NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

EMPLOYING DECEPTIVE DYNAMIC NETWORK TOPOLOGY THROUGH SOFTWARE-DEFINED NETWORKING

by

Jason J. Hughes

March 2014

Thesis Advisor:	Robert Beverly
Second Reader:	Frank Krautheim

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	03-28-2014	Master's Thesis 2012-09-24 to 2014-03-28

4. TITLE AND SUBTITLE

EMPLOYING DECEPTIVE DYNAMIC NETWORK TOPOLOGY THROUGH SOFTWARE-DEFINED NETWORKING

5. FUNDING NUMBERS

N6600112WX01357

6. AUTHOR(S)

Jason J. Hughes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DHS, Science and Technology Directorate
Washington, DC 20528

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Computer networks are constantly being actively probed in attempts to build topological maps of intermediate nodes and discover endpoints, either for academic research or nefarious schemes. While some networks employ recommended conventional countermeasures to simply block such probing at the boundary or shunt such traffic to honey pot systems, other networks remain completely open either by design or neglect. Our research builds on previous work on the concept of presenting a deceptive network topology, which goes beyond conventional network security countermeasures of detecting and blocking network probe traffic. By employing the technologies from the emerging field of Software-Defined Networking and the OpenFlow protocol, we constructed a custom-built SDN controller to listen for network probes and craft customized deceptive replies to those probes. Through employment of various network probing utilities against our custom-built SDN controller in a test network environment, we are able to present a believable deceptive representation of the network topology to an adversary. Therefore, this work demonstrates that the primitives of the expanding OpenFlow protocol show strong potential for constructing an enterprise-grade dynamic deceptive network topology solution to protect computer networks.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Network mapping, Software-Defined Networking (SDN), OpenFlow, Network Security, Computer Network Defense (CND), Topological Deception	111
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

EMPLOYING DECEPTIVE DYNAMIC NETWORK TOPOLOGY THROUGH SOFTWARE-DEFINED NETWORKING

Jason J. Hughes
Lieutenant, United States Navy
B.S., Hawaii Pacific University, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN CYBER SYSTEMS AND OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
March 2014

Author: Jason J. Hughes

Approved by: Robert Beverly
Thesis Advisor

Frank Krautheim
Second Reader

Cynthia Irvine
Chair, Cyber Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Computer networks are constantly being actively probed in attempts to build topological maps of intermediate nodes and discover endpoints, either for academic research or nefarious schemes. While some networks employ recommended conventional countermeasures to simply block such probing at the boundary or shunt such traffic to honey pot systems, other networks remain completely open either by design or neglect. Our research builds on previous work on the concept of presenting a deceptive network topology, which goes beyond conventional network security countermeasures of detecting and blocking network probe traffic. By employing the technologies from the emerging field of Software-Defined Networking and the OpenFlow protocol, we constructed a custom-built SDN controller to listen for network probes and craft customized deceptive replies to those probes. Through employment of various network probing utilities against our custom-built SDN controller in a test network environment, we are able to present a believable deceptive representation of the network topology to an adversary. Therefore, this work demonstrates that the primitives of the expanding OpenFlow protocol show strong potential for constructing an enterprise-grade dynamic deceptive network topology solution to protect computer networks.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACL	Access Control List
API	Application Programming Interface
APT	Advanced Persistent Threat
ARP	Address Resolution Protocol
ARPANET	Advanced Research Project Agency Network
CAIDA	Cooperative Association for Internet Data Analysis
CAN	Campus Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DARPA	Defense Advanced Research Project Agency
DNI	Director of National Intelligence
DNS	Domain Name System
DOD	Department of Defense
DoS	Denial of Service
DSCP	Differentiated Services Code Point
ECN	Explicit Congestion Notification
FOSS	free and open-source software
FQDN	Fully Qualified Domain Name
GNS3	Graphic Network Simulator
GUI	Graphical User Interface

IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IOS	Internetwork Operating System
IP	Internet Protocol
IPID	Internet Protocol Identification
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPS	Intrusion Prevention System
ISI	Information Sciences Institute
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
MSTD	Moving Standard Deviation
NIC	Network Interface Card
NPS	Naval Postgraduate School
NSA	National Security Agency
NSM	Network Security Monitoring
NFV	Network Foundations Virtualization
OF	OpenFlow

ONF	Open Networking Foundation
OOBM	out-of-band management
OSI	Open Systems Interconnect
OXM	OpenFlow Extensible Match
POX	Python OpenFlow Controller
QDR	Quadrennial Defense Review
QoS	Quality of Service
RFC	Request For Comments
RIP	Routing Information Protocol
RIPE NCC	Reseaux IP Europeans Network Coordination Center
RIR	Regional Internet Registries
RTT	Round-trip Time
SANE	Secure Architecture for the Networked Enterprise
SDN	Software-Defined Networking
SFD	Start Frame Delimiter
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TLV	Type-Length-value
ToS	Type-of-Service
TTL	Time-to-Live
TTP	Tactics, Techniques, and Procedures

UDP	User Datagram Protocol
USB	Universal Serial Bus
USC	University of Southern California
USCYBERCOM	United States Cyber Command
USSTRATCOM	United States Strategic Command
VLAN	Virtual Local Area Network
VNS	Virtual Network System

Acknowledgements

First and foremost, I would like to thank my wife for all her support in my pursuing completion of a master's degree. I would also like to acknowledge the many professors and lecturers at Naval Postgraduate School (NPS) for which I have gained insightful knowledge either through attending courses, ad-hoc discussions, embarking on off-campus visits, and the extensive list of excellent guest lectures of which I was privy to sit in on. I would like to thank James "Murphy" McCauley, one of the primary developers and supporters of the Python OpenFlow Controller (POX) for his patience and assistance answering my many questions on the POX Wiki as I stumbled in learning the OpenFlow (OF) protocol and the POX library in creating a custom POX controller for this thesis work. Lastly, I would especially like to thank my thesis advisor, Professor Robert Beverly, and my second reader, Dr. Frank (John) Krautheim, for their extensive professional knowledge, and their continued direction, guidance, patience, and support during this thesis research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: INTRODUCTION

In this chapter, we discuss the motivation that led to this research with a quick introduction on network mapping and the OpenFlow (OF) protocol. We will discuss the value this research has to the Department of Defense (DOD) in the realm of network security and the intended contributions of this research to previous work on using a deceptive dynamic network topology in defending computer networks.

1.1 Motivation

Computer networks have become such an integral part in all aspects of our life today, enabling the rapid globalization of peoples and societies. They have changed the way we communicate and share information, get local, national, and international news, entertainment, run governments, manage economies, banking, and businesses, to the way we gather intelligence and militaries wage war. With computer networks and the Internet, we have seen the rise of entire industries that seemed inconceivable only a few decades ago, while other industries are have been completely transformed.

It is difficult to imagine how we lived without these technological advances that computer networks have brought to our lives and even more difficult to imagine how we could survive if these networks were brought down. As such, we continually study the various aspects of computer networks and networking technologies in search for ways to improve understanding, reliability and security of networks, and in turn develop new technologies and techniques for constructing more efficient and secure systems. This work herein will continue that endeavor, exploring developing technologies to evaluate their potential application in better protecting computer networks.

1.1.1 Mapping

The Internet has become so complex that entire research groups have been formed with a focus of conducting internet measurement [1], [2], and entire conferences dedicated to the presentation and sharing of their research and findings [3]. To aid in this research, evermore intelligent and complex tools and methods are developed to perform measurements.

One measurement goal is creating intelligent topological maps and visualization of the Internet and computer networks. This activity comes from a wide range of actors on the Internet beyond just reconnaissance and those conducting mapping in the name of harmless research [4]. As computer networks continually come under more advanced methods of probing and attack, we need to explore new and innovative ways to detect and counter these infiltration and exploitation attempts. We will discuss in further detail network mapping in §2.1, the methods and tools used for network mapping in §2.2, and ways that mapping techniques can be deceived in §2.3.

1.1.2 OpenFlow

With the development of more advanced applications and mobile computing, to computer and network virtualization, deployment of advanced security appliances like an Intrusion Detection System (IDS) and Intrusion Prevention System (IPS), Network Firewalls, integration of intelligent Network Security Monitoring (NSM) solutions, and the rapid explosion of cloud computing, much has changed in the computer industry. But all this growth and change in years past has restricted network and security experts to implementation of closed, vendor-specific solutions to best ensure interoperability across the enterprise. Though significant work has been done in improving traffic management methods at the lower layers of the Open Systems Interconnect (OSI) model, with more advanced routing and switching protocols that span vendors, current network architectures are still limited by technologies the equipment providers choose to support.

The emerging field of Software-Defined Networking (SDN) [5] and the OpenFlow (OF) [6] protocol are enabling the development of innovative network architectures and solutions that are completely vendor agnostic and custom tailored for the information systems they connect. International Data Corporation (IDC), a premier global market analysis firm, in their published predictions for 2013 had estimated that the Software-Defined Networking (SDN) market will reach $3.7 billion by the year 2016 and account for over 35 percent of Ethernet switching in datacenters [7]. SDN has become so disruptive to the networking industry in the past few years that most of the big named network equipment providers are rapidly working to release OF-enabled products into the market, and in turn define and promote their own vision for development of SDN solutions [5].

The Open Networking Foundation (ONF), a user-driven organization dedicated to the promotion and adoption of SDN, oversees the maintenance and development of the OF protocol standards, and maintains a dynamic list of vendors and products that support the OF protocol [5]. The primitives of SDN and specifically the OF protocol, which we will explore further in §3, as defined in an ONF White Paper [8], are:

1. Decoupling of the network switch management control plane from the data plane.
2. Centralized controller to manage one to many switches.
3. Ability to manage the behavior of the network using well-defined interfaces and standards based program.
4. Ability to manage devices across multiple vendor switches through non-proprietary programmability.

1.2 Value to the Department of Defense

Open and democratic societies, such as the United States, are connected and dependent on the free flow of information though computer networks for daily life and public safety. And as such, they are critically vulnerable to the potential for disruption to the various infrastructure upon with it operates. Therefore, it is of little surprise that issues in Cyberspace have quickly risen to the forefront of strategic thinking and policy in the last few years. The*United States National Security Strategy* [9], signed by the president in May 2010, highlights Securing Cyberspace as an important element in our strategic approach to pursuing our four enduring national interest: Security, Prosperity, Values, and International Order. Specifically with regard to security, the national strategy stated that "Cybersecurity threats represent one of the most serious national security, public safety, and economic challenges we face as a nation." It also recognizes cyber as a fifth domain by which the military must continue to have capabilities to operate within in defense of the country and our allies.

The DOD understands the strategic significance of protecting against cybersecurity threats given the establishment of the United States Cyber Command (USCYBERCOM) in 2009; a sub-unified command under the United States Strategic Command (USSTRATCOM). General Keith Alexander, USA, as then Director of the National Security Agency (NSA), became dual-hatted as the first commander of USCYBERCOM, whose mission would

be focused on operation and defensive of networks within the .MIL domain. Given that DOD networks are constantly being probed and scanned millions of times a day, USCYBERCOM's mission will be no small task [10].

The Quadrennial Defense Review (QDR) Report [11], released in February 2010, by former Defense Secretary Robert Gates, highlights that U.S. deterrent capabilities remains grounded in land, air, and naval forces, but that these forces are enabled by cyber and space capabilities. The report also recognizes that the security environment requires cyber defense capabilities and list several steps the DOD was taking to strengthen those capabilities. Centralization of cyber operations under USCYBERCOM was one of these major steps. Its establishment better positions the DOD to also address one of six identified missions from the QDR that the DOD must continue to focus on improving policy, doctrine, and capabilities within; to "operate effectively in cyberspace."

Former Defense Secretary Leon Panetta, like his predecessor Robert Gates, understood the significant cybersecurity threat to the nation during his time in office. Panetta was quoted towards the end of his tenure that "there is no question, in my mind, that part and parcel of any attack on this country in the future, by any enemy, is going to include a cyber element" [12]. Just after taking office in July 2011, Panetta released the *Department of Defense Strategy for Operating in Cyberspace* [10] in response to the 2010 QDR. This report further described the departments' reliance on computer networks, and it established five strategic initiatives as a roadmap for the DOD:

1. Treat cyberspace as an operational domain to organize, train, and equip so that DOD can take full advantage of cyberspace's potential.
2. Employ new defense operating concepts to protect DOD networks and systems.
3. Partner with other U.S. government departments and agencies and the private sector to enable a whole-of-government cybersecurity strategy.
4. Build robust relationships with U.S. allies and international partners to strengthen collective cybersecurity.
5. Leverage the nation's ingenuity through an exceptional cyber workforce and rapid technological innovation.

Taking further steps to enable the U.S. military to operate effectively in cyberspace, the Joint Chiefs released JP 3-12, *Joint Cyberspace Operations* [13], in February 2013. Due to the potential sensitivity of offensive and defensive operations in cyberspace, the publication was classified as SECRET and only available to cleared U.S. persons and specific allied partners with proper need-to-know. Only weeks after the publication of JP 3-12, the Director of National Intelligence (DNI), James Clapper, presented the *Worldwide Threat Assessment of the US Intelligence Community* [14] to the Senate Select Committee on Intelligence. He opened the brief on the increased global threat in cyber, highlighting several recent cyber attacks of a nation-state acting against another and ongoing cyber espionage; which if reported in the public domain are tracked by the Center for Strategic and International Studies in *Significant Cyber Incidents Since 2006* [15]. The director discussed the growing concern to U.S. critical infrastructure and that a remote chance existed that an attack could be waged against the U.S., not from advanced cyber actors such as Russia or China, but more likely from an isolated, but capable, state or non-state actor.

The previous work on deceptive topology [16] we are building on here already spoke at length regarding the history and use of mlitary deception and its application in cyberspace. With the ever-growing complexity of computer networks, we have often taken a Defense-in-Depth approach in which no one single control has proven sufficient to protect our infrastructure. The DNI also stated in his report that "in some cases, the world is applying digital technologies faster than our ability to understand the security implications and mitigate potential risks" [14]. Therefore, in support of the DOD's five strategic initiatives as previously stated, specifically "employing new defense operating concepts to protect DOD networks and systems" and "leverage the nation's ingenuity through rapid technology innovation," it is important that the department explore the potential of new ideas and technologies in support of cyber defense and defense of this nation. Deploying a deceptive dynamic network topology at the major DOD ingress points could further enhance the defense posture of DOD networks from potential adversaries.

1.3 Summary of Contributions

Previous research [16] was conducted in employing various deceptive techniques, based on the concepts of military deception, in preventing an adversary from successfully mapping the true topology of the network. This work represents continuing research to present al-

ternate methods to more optimality employ deceptive topology in real-world networks. We will explore the growing field of SDN and specifically the OF protocol to suggest a proof-of-concept enterprise solution to leveraging deceptive topology. Then we will demonstrate that the primitives of the OF protocol provide a more agile approach to employing topology deception through a custom-built SDN network controller. To validate the believability of our deceptive topology solution, we will utilize several common network mapping tools to probe our test network, running our custom-built controller, in attempts to generate a map of the network topology. We then present our findings and discuss some limitations of the current OF standard, and suggest possible changes to the OF protocol which could better enable a more robust OF topology deceptive controller.

The remainder of this thesis is organized as follows. In Chapter 2, we will further explore network mapping and the tools and techniques employed. In Chapter 3, we will explore and in-depth background of the OF protocol. In Chapter 4, we discuss the details of the topological deception implemented through the use of the OF protocol. In Chapter 5, we will present the findings from our experimentation. Finally, in Chapter 6 we will provide our conclusion and suggest future work.

CHAPTER 2:
BACKGROUND

To understand the growing need to better secure computer networks and the desire to obfuscate their topology, we must explore further the current efforts to map those networks, and to what ends. We will discuss current research mapping methods comparatively with efforts to map networks for exploitation. We will briefly review the underlying protocols that support mapping and some of the available tools employed. We will then explore some of the challenges in network mapping that can cause unintended representations of network maps. Last, we will review some technologies that can disrupt network mapping efforts and review previous work toward purposefully presenting a deceptive network topology.

2.1 Mapping

Computer networks are constantly being actively probed in the effort to generate a topological map of their existence. A map of a networks' topology is a depiction of the networks' physical or logical structure. From a physical perspective, it shows the arrangement of various intermediate and endpoint nodes, such as routers, switches, proxies, firewalls, servers, just to name a few, and how they are interconnected, or linked together. From a logical perspective, it shows how data and information flows in and through the network, regardless of its physical structure. This reconnaissance traffic to build topological maps emanates from many sources to include research groups and curious learners, to more nefarious purposes from a wide range of hacker activity, from the simple script kiddie, to more experienced criminal hackers, and the nation-state Advanced Persistent Threat (APT).

2.1.1 Mapping for Research

The Internet as we know it today began as the Advanced Research Project Agency Network (ARPANET), a research project in studying packet switched networks in the 1960s funded by the Defense Advanced Research Project Agency (DARPA). It was an attempt to develop a replacement to circuit-switched networks for the movement and sharing of data, with many of the first nodes located at some of the major universities for which the DOD funded various other research. One of the key underlying technical ideas that live on in the Internet

of today was open architecture networking. In its early years, the Internet was mostly available to research organizations and universities, but began rapid expansion in the 1990s with the privatization of providers and commercialization of network products [17].

The growth of the Internet over the last two decades has exploded: hundreds of countries are connected and over 34 percent of the world's population is now online [18]. The Internet, being a network of networks, is now quite complex and constantly changing and evolving by the minute, with individual nodes to entire networks coming online or going offline, or even moving between networks. Understanding the Internet's states and reason for changes is certainly of interest to the research community, and the various public and private groups that fund their research.

Much of the Internet at its core is generally constructed through interconnections, or peering, between major communications providers, or tier 1 Internet Service Providers (ISPs). Since these major providers typically do not pay each other for the peering connection, it is cost advantageous for a tier 1 provider to hand off traffic that is not destined to a node or network within its own network, to another tier 1 provider's network. Therefore, it is interesting to understand, through network mapping, how and where these major providers establish peering, and how they route traffic between themselves. These tier 1 ISPs usually connect smaller tier 2 and tier 3 ISPs, who in turn connect public and private customer networks through customer-provider links. In customer-provider links, the customer pays the provider for the connection, be it a public or private network to an ISP, or a higher tier ISP to a lower-tier ISP. Interestingly, we are learning through network mapping, that more and more upper tier providers and major public and private networks are establishing mutual peering between themselves, lowering their cost by taking out the middle man: the lower-tier ISPs [19].

In the United States, we believe in the right to free speech as established by our Constitution. This extends as well to our activities on the Internet where we enjoy the idea of openness and the unrestricted sharing of ideas and information. However, in some non-democratic nation-states, governments may seek to restrict or completely block Internet access to its citizens. A primary example is China's Great Firewall [20] or Iran's attempts to build a national Internet [21], or various nations' attempts to close and limit Internet access during the Arab Spring in 2011, as was done in Egypt and Libya [22]. Such con-

ditions of governmental restriction and political unrest, where the people have some form of network technology or the Internet by which to communicate, coordinate, and virtually assemble, will naturally be an area of interest for researchers of more democratic nations.

The Cooperative Association for Internet Data Analysis (CAIDA) is an organization dedicated to "investigating practical and theoretical aspects of the Internet" and "provide macroscopic insights into Internet infrastructure, behavior, usage, and evolution" [1]. At the heart of supporting this measurement research is the Archipelago Measurement Infrastructure [23], or Ark, with approximately 80 monitors deployed worldwide in the beginning of 2014. This distributed measurement architecture provides researchers the ability to perform various on-demand topology measurements from geographically distributed vantage points on the Internet. CAIDA makes their data freely and publicly available to researchers for download, performing anonymization of the data when necessary.

Another large research project in building Internet measurement infrastructure is from the Reseaux IP Europeans Network Coordination Center (RIPE NCC) [24]; one of the five Regional Internet Registriess (RIRs) that provide Internet resource allocations and registration services worldwide. RIPE NCC is the RIR for Europe, but also covers the Middle East and Central Asia. The RIPE Atlas [25] project consists of over 4700 probes as of early 2014 and aims to be the largest Internet measurement infrastructure. The vantage point nodes are a tiny hardware device, about the size of an external Universal Serial Bus (USB) hub, deployed on an Internet connected Local Area Network (LAN) and capable of performing measurements such as connectivity and reachability tests. Of the currently deployed probes, the largest percentage of total probes, over 12 percent, are hosted in the United States. Vice coordinating partnered deployment of hosts at various sites around the world as done with CAIDA's Ark, RIPE Atlas probes are requested by users interested in Internet performance and are frequently hosted on their home Internet connections. The specific measurement toolset installed on each probe is configured and controlled by RIPE NCC, which restricts the scope and type of scans that can be performed.

A global research project more attuned to providing a platform for testing large-scale distributed network services is PlanetLab [26]. With 1175 nodes at 564 sites worldwide as of the beginning of 2014, PlanetLab presents researchers with a unique ability to test new technologies in a real world environment. Like CAIDA, most of the PlanetLab nodes are

hosted at research institutions, with some co-located at major network routing centers. All nodes run a common Linux operating system, with additional support software, to enable researchers to get a "slice" of one or more nodes for experimentation. In this fashion, PlanetLab is able to support many different experiments running side-by-side on the nodes, but in isolation of one another. Where CAIDA's Ark and RIPE Atlas are in different ways restricted in the employment of their nodes, researchers can build a customized network measurement scanner, upload it to their PlanetLab slices, and have complete control to manipulate the tool and perform probes and scans. However, individuals cannot just go online and request slices in PlanetLab, upload some custom-developed scanner, and start scanning the Internet. To use PlanetLab, and individual must be associated with an academic, industrial, or government institution that is a member of the PlanetLab Consortium.

The three projects mentioned, CAIDA's Ark, RIPE Atlas, and PlanetLab slices, all to different degrees, with different requirements and restrictions, enable measurement of intermediate nodes (routers, proxies, etc.), down to end point nodes (hosts, servers), in building topological maps. There are certainly others conducting various topology mapping projects to discover nodes and endpoints through networks, but some mapping projects are focused just on mapping the end points themselves. One such project was the *Ant Censuses of the Internet Address Space* [27], a project by researchers at the Information Sciences Institute (ISI), a unit of University of Southern California (USC), which focused only on discovering and mapping end hosts in efforts to determine the density of the Internet Protocol version 4 (IPv4) address space. In this project, the researchers probed for the presence of each individual Internet Protocol (IP) address in the entire IPv4 address space, recording the results of which hosts directly responded, were reported as unreachable, or simply did not respond at all. These early measurements were conducted over a four-year period from 2003 to 2007 [28]. Continued work under the Ant project from 2007 to 2011, employed additional methods of constructing two-dimensional color coded maps of the results, which allowed users to drill down into subsets of the IP space [27].

Another less-sanctioned "Internet Census" was performed by unnamed researcher(s) in 2012, called the Carna Botnet [29]. In this work, they searched the Internet IPv4 address range for network devices with a few default logon credentials, such as root:root, or admin:admin. From the list of discovered devices, they determined which ones would accept

a small payload, which supposedly would not impede the primary functions of the host, and installed a custom scanner to run at a low level priority. The researchers claimed to have had at most 420,000 devices running in their botnet, by which they performed various probes and scans of the Internet. From the results, the researchers constructed a two-dimensional map of the IPv4 address space, along with other generated statistics, similar to that performed by the Ant Census project.

As we have shown, there is an extensive and diverse amount of activity performed on the Internet in efforts to build topology and end-point maps of networks for various research projects. There are many more curious individuals with an Internet connection and a desire to learn performing probes and scans across the IP address space as well, all in an effort to improve their own individual knowledge. When considering that DOD networks are reportedly probed millions of times a day, it is difficult, if not impossible, to distinguish whether this probing activity is simply for research and not for nefarious purposes.

2.1.2 Mapping for Exploitation

Hacking and exploiting a system is generally the act of gaining unauthorized access to some computer system and exploiting it for your own purposes. Though this could again be just the curious learner who has crossed the line of legality, knowingly or unknowingly. The purpose could also be much more nefarious in nature to the point of criminal behavior, or worse, depending on the actions and actual intent of those who gained the access into a system.

Hackers are generally classified as either a White Hat or a Black Hat, though many may operate somewhere between these classifications [30]. We generally consider a White Hat hacker to be a security experts who are hired to perform authorized penetration testing of a network to uncover vulnerabilities and exploit them, then report their findings based on the signed Penetration Agreement with the organization for which they were authorized. This is more in tune to the original definition of what it was meant to be a hacker. On the contrary, a Black Hat hacker is one who also uncovers vulnerabilities and exploits them, but lacks authorization to do so, fails to inform the network or system owner of the vulnerability, and may even take steps to share the information with other Black Hats. The actions of a Black Hat are what people have more generally come to associate with the

term hacker and hacking today; the malicious hacker or cracker. Both of these "hackers," White Hats and Black Hats, require similar skill sets and use many of the same tools to perform their actions of hacking and exploiting: the clear distinction is the purpose and intent behind the acts.

Whether it is a White Hat or a Black Hat attempting to discover vulnerabilities and exploit them, the general methodology [31] each follows is the same. The initial steps in the process are commonly referred to as footprinting and scanning, though footprinting can also involve some forms of scanning. Anyone who was going to plan a robbery with the intent of minimizing the chances of getting caught, would generally attempt to find out as much as they could about the target, such as entry and exit points, security controls and systems, and many other relevant attributes about the place. A similar process is applied in attempting to gain unauthorized access to a computer network. There is a plethora of passive data collection that must first be conducted by reviewing publicly available information to construct a profile of the organization's Internet presence. This information can be obtained from the target organization's websites and by reviewing publicly accessible RIR information. Through these resources we can learn the organization's structure, partners, as well as domain names, IP network blocks, other public facing Internet services and their associated IP addresses, to include information about their security posture.

Armed with the information gained from passive footprinting a target, specifically now a list of IP addresses and network blocks to focus attention to, the steps of active footprinting and scanning are performed. At this point, the goal is to probe the list of addresses to discover network routers, servers, and other intermediate and end point nodes within the target network. Essentially, to develop a general topological map of the network, its links, and discover weaknesses to exploit. The goal of the hacker could be to perform a Denial of Service (DoS), by taking down a critical node or link, or exploit a node to gain further access into the network. If an adversary conducting the probes and scans is somewhat cautious and methodical in their approach to active footprinting and scanning, from a network defense perspective, the traffic may not look much different than the type of traffic seen from activities discussed in §2.1.1 on research mapping.

2.2 Mapping Tools

The foundation that supports the various methods of network mapping, whether for research, network troubleshooting, or more nefarious purposes, is the inherent protocols within the Transmission Control Protocol/Internet Protocol (TCP/IP) [32] suite. The TCP/IP protocol suite is the defacto standard required to be run by any computer to participate in communications within a computer network, whether a small LAN or the Internet. We will discuss briefly some of these protocols, how they work, then discuss some of the various tools that utilize features of these protocols.

2.2.1 Internet Control Message Protocol

Sending messages between computers is handled by the IP portion of TCP/IP; however, there are various ways in which errors in the communication between host can occur. This is in part because IP contains no method to ensure reliable communication between host as this is left to upper and lower layers of TCP/IP, whether end-to-end or host-to-host communication. The Internet Control Message Protocol (ICMP) [33] is an integrated part of IP to communicate various error messages between hosts regarding the delivery of individual datagrams. An ICMP message is generated based on the error condition observed by a network router or individual host computer. One of the most common situations in which an ICMP message may be generated is when a datagram cannot reach its destination. The error message is packaged as the payload within an IP datagram.

An ICMP message contains as a basic header to differentiate various error messages, a Type and Code field, along with a calculated checksum. Some ICMP message Types have various associated Codes to further describe the error type, while other Types do not use a Code value. As mentioned, one of the most common situations in the generation of an ICMP error message is when a datagram cannot reach its destination, which is a Type 3 “Destination Unreachable” message. A Type 3 message has fourteen different options for the associated Code to further articulate the specific reason why the destination was unreachable. Some of these Codes are specifically generated from a router, such as Code 1 for Host Unreachable, and some specifically generated by a host computer, such as Code 3 for Port Unreachable. There are several other common ICMP Types exploited by many of the various probing and scanning tools: Type 8 for Echo, Type 0 for Echo Reply, and Type 11 for Time Exceeded.

2.2.2 Transport Protocols

The primary purpose of TCP/IP, however, is not to move ICMP error messages around, but instead to move, or transport, various types of data packets between computers in a network or across the Internet. These various data packets can be simple data files, electronic mail, to streaming audio or video. As such, there are two transport protocols that sit on top of IP to control the movement of data between two computers: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [32]. Each has its pros and cons, but overall each is designed to move information from a program on one computer to a program on another computer. Each program, such as an email server or a web server, communicates through a port number, which separates one service from another service running on a computer. Both TCP and UDP make use of port numbers in the transport header, a separate 2^{16} value field for source port and for destination port, which can range from zero to 65535. Some common, or well known, port numbers are 25 for a Mail Server and 80 for a web server. Some network scanning tools make use of specially crafting TCP and UDP packets in order to conduct probing and scanning as legitimate network services also make use of these protocols and ports.

Transmission Control Protocol

TCP is a connection-oriented transport protocol for end-to-end data communications. It is connection-oriented in that it uses Flags in the header to establish, manage, and tare down sessions between two communicating computers in a controlled fashion. It also uses sequencing and acknowledgment fields in the header to control and guarantee delivery of each and every transmitted packet and employs methods to resend packets that fail to reach the far end. As well, it employs various windowing and options to control the flow of information. TCP is typically employed where precise reconstruction of information from one end to the other is required, as is the case with moving a data file from one location to another.

User Datagram Protocol

UDP is a connectionless transport protocol for end-to-end data communications. It is connectionless in that it minimizes overhead and speeds processing of packets to the end host by sacrificing the use of end-to-end delivery guarantees that TCP utilizes. As such, UDP does not contain sequencing, acknowledgment, flags, windowing, or options fields in the

header as does TCP. UDP is typically employed where precise reconstruction of information from one end to the other is not required and some loss is acceptable, such as streaming audio or video. Otherwise, UDP leaves it up to the application to decide how to handle lost packets, such is typical done with Domain Name System (DNS) queries.

2.2.3 PING Utilities

PING [34] is a network probing utility to perform network diagnostics by querying a remote host to determine if that host is alive and communicating on the network. The utility makes use of particular ICMP message Types as discussed in §2.2.1. To probe a host, PING constructs an ICMP Echo message with Type 8, Code 0 set, as well as an identifier and sequence number field of the ICMP header, and addresses it to a particular destination IP address as indicated in the IP header portion of the packet. If the destination host receives the Echo message, it would normally construct an ICMP Echo Reply message with Type 0, Code 0 set in the ICMP header, and address it back to the host that sent the Echo message. PING calculates a Round-trip Time (RTT) for each probe message, which is the time difference between when a particular probe was sent and when its associated reply was received. Much of this RTT is due to inherent propagation delay as the Echo packet is transmitted along some medium to the target and the Echo reply packet is transmitted back along that medium to the source. However, other factors can affect the RTT by further causing delays in packets or even packet loss. PING uses the identifier and sequence number of each probe packet as a unique signature to associate reply messages. If, for whatever reason, along the network path the Echo message cannot be delivered to the destination address, the last node to receive the Echo message may generate a destination unreachable, Type 3, with the appropriate Code set, and address it back to the source as identified in the original Echo message. As well, it is possible that a particular probe may never receive a reply, which then results in packet loss.

In the example demonstrated in Figure 2.1, we sent four probe messages to an IP address of 172.20.5.2, and in this case received a reply to each probe. Each line of "64 bytes from..." represents a reply to an individual probe and includes the calculated RTT for each probe. In the ping statistics, we are presented the details of probes sent and responses received, and the calculated packet loss. Since we received an Echo reply to each Echo probe message, our calculated packet loss is zero. The statistics also present some valuable information

with regard to overall RTTs to include the minimum, average, and maximum return times. The last value in the RTT statistics line is the calculated mdev, also known as the Moving Standard Deviation (MSTD), which is an average of how far each ping RTT is from the mean RTT. The higher the mdev, the more variable the RTTs are over time. Though it is unlikely that the mdev would be zero, the lower the mdev number the better. These latency calculations can be extremely useful in topology mapping through various latency-based geolocation methods [35], [36], [37].

```
PING 172.20.5.2 (172.20.5.2) 56(84) bytes of data.
64 bytes from 172.20.5.2: icmp_seq=1 ttl=59 time=104 ms
64 bytes from 172.20.5.2: icmp_seq=2 ttl=59 time=105 ms
64 bytes from 172.20.5.2: icmp_seq=3 ttl=59 time=101 ms
64 bytes from 172.20.5.2: icmp_seq=4 ttl=59 time=100 ms

--- 172.20.5.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 100.403/103.019/105.102/2.072 ms
```

Figure 2.1: Demonstration of successful PING to remote host.

The basic PING utility first developed in the 1980s employed ICMP and was initially installed on Berkley Unix operating system. Some of the natively installed PING utilities embedded in operating systems are restricted in their functionality and do not offer a wide range of options, nor do they integrate well with custom scripts. One major limitation is the lack of ability to ping more than one IP address simultaneously, for which the Fping [38] utility was developed. Fping provides various methods to specify and ping multiple hosts at the same time, while also providing improved scripting functionality for integration into custom applications.

Some Internet routers may not respond to an ICMP Echo message if the owners of that router strictly implements Request For Comments (RFC) 792 for ICMP, in that "no ICMP messages are sent about ICMP messages" [33]. However, RFC 1122 [39] later provided clarification on specific ICMP Types and situations in which a router should and should not respond to various ICMP messages, and it specifically stated that routers are to send Type 11 Time Exceeded messages in response to Type 8 Echo messages. On the other hand, network security personnel over the years have implemented controls to block ICMP into private networks per various security best practices. As an example, *The 60 Minute Network Security Guide* [40] published by the NSA, recommends restricting certain ICMP

message Types. Specifically, it recommend only allowing the following ICMP Types into a private network: 0 - Echo Reply, 3 - Destination unreachable, 4 - Source Quench, 11 - Time Exceeded, and 12 - Parameter problem; all of which would normally be triggered and generated by traffic that originates from internal of the private network. Additionally, it recommend only allowing the following Types out of a private network: 4 - Source Quench, 8 - Echo Request (PING), and 12 - Parameter Problem; all of which supports troubleshooting from internal of the network. The other ICMP Types from the RFC not specifically allowed, should be blocked. The implementation of such network security best practices, of blocking ICMP at a private networks' border router, can cause ICMP based pings and traceroutes to fail.

Given these practices of restricting ICMP traffic, other popular ping utilities were developed to overcome these security controls. One popular utility is Hping [41], which allows probing a host not only with ICMP, but also using common communication transport protocol messages described in §2.2.2. Using Hping to conduct a UDP or TCP probe for a host on a commonly open public facing port, may allow discovery of nodes that otherwise would not be discoverable with an ICMP probe. Another utility that started as a "Google Summer of Code" Project in 2009 is Nping [42], which allows full control over protocol headers for network penetration and stress testing. A unique feature of Nping is an "Echo mode" for advanced troubleshooting and discovery so users can see how packets change in transit without the use of packet capture utilities for in-depth packet comparison.

2.2.4 Traceroute Utilities

Traceroute is a network probing utility designed for network diagnostics to determine the route, or hops, along a network or the Internet, to a destination node. That destination node can be a router port, end host computer, or some other networking device on the network. It works somewhat the same as the various PING utilities mentioned in §2.2.3, in that it records associated return packets and calculates a RTT, but traceroute also manipulates the Time-to-Live (TTL) field of an IP header for each probe. The IP header of a packet includes a TTL value to prevent packets from looping endlessly on a network or the Internet. When a network router receives an incoming packet on its interface, it will first inspect the packets' TTL, and if that value is a 1 or 0, the router will drop the packet and construct an ICMP Type 11, Code 0, Time Exceeded / TTL Exceeded in Transit message, and send it back to

the source of the dropped packet. Typically the source address of the router generated Type 11 message will be the address assigned to the router ingress port that received the packet, which in turn allows traceroute to reconstruct a forward interface-level path to the target. This is contrary to the seldom supported "Record Route" option of the PING utility, which records the egress, or outbound port, of each router along the path to the target as discussed in RFC 791 [43].

When running most traceroute utilities [44], it will by default send three probes at each TTL increment, starting with a TTL value of 1. In other words, three packets with a TTL value of 1 will be sent before the probing computer increments the TTL by 1 for the next set of three probes, this time with a TTL of 2. With each individual probe message, traceroute will also increment the Identification number in the IP header and the Sequence number in the ICMP header, which uniquely identifies each probe message. As increasing TTL probes are sent, their packets TTL will be decremented by the routers along the path to the target, by which the prober will receive associated Time Exceeded messages and infer a forward path to the target. The traceroute utility will continue to send each set of probes until a response is received by the destination target or the maximum number of hops to measure is reached. Not all traceroute utilities exactly follow this method as path anomalies, discussed further in §2.3.1, can arise when probing across certain types of network architectures, by which alternate utilities and methods are used to overcome these anomalies.

An example of a successful ICMP traceroute is illustrated in Figure 2.2. Here we see each output line starts with the tested TTL value in succession until the destination was reached. Each line lists the IP address of the returned probe at that TTL value, along with the calculated RTT of the individual probes at that TTL. The default for most traceroute utilities again is three probes per TTL, hence why three different RTTs are listed per TTL. We can also see that, by default, the traceroute utility will only record up to 30 hops max towards the target before giving up. Some utilities may employ a gap limit, stopping the probe after a certain number of failed replies in succession, and hence not reach the maximum hop limit. Utilities that employ gap limits will typically allow options to modify the gap limit of a trace.

The first traceroute program used ICMP to perform the probe messages, however, vendor interpretations of various early RFCs on how to handle ICMP traffic, as mentioned in

```
traceroute to 172.20.5.2 (172.20.5.2), 30 hops max, 60 byte packets
 1  172.20.9.1  4.162 ms  10.003 ms  11.687 ms
 2  172.20.1.2  32.017 ms  38.620 ms  32.175 ms
 3  172.20.2.2  52.555 ms  51.489 ms  55.265 ms
 4  172.20.3.2  70.271 ms  77.426 ms  73.474 ms
 5  172.20.4.2  92.828 ms  92.867 ms  91.616 ms
 6  172.20.5.2  112.653 ms  102.199 ms  106.639 ms
```

Figure 2.2: Demonstration of successful ICMP traceroute to remote host.

§2.2.3, caused erratic performance with this method. As such, researchers sought alternate methods, such as sending UDP probes with incremental TTL values, to solicit Time Exceeded messages from network routers along the path to the target. Since this method is employing a transport protocol typically used for normal network traffic, a port number as discussed in §2.2.2 must be assigned. The original implementation developed by Van Jacobson and still in use today, employs UDP ports 33434 through 33534, which was a high enough port that at the time no hosts would be using. Since this method directs traffic to a port that no host is expected to be listening on, the traceroute expects to receive a different type of ICMP reply message than that from an ICMP traceroute. Specifically, the end host should reply with an Type 3, Code 3, Destination Unreachable / Port Unreachable ICMP message [44].

Like the PING utility, the traceroute utility is embedded in practically all operating systems today, however, their default behavior differs by operating system. Specifically, Windows operating systems employ the ICMP traceroute method, while versions of the Linux and Unix operating systems employ the UDP traceroute method. In all other aspects, the utilities are somewhat basic with limited selectable options; though many Linux flavors will include additional options for conducting ICMP and even TCP traces [44]. There are many other third party traceroute utilities, such as the popular paris-traceroute utility [45], that employs various methods of tweaking header information in ICMP, UDP and TCP traces to more accurately map a network path.

2.2.5 Other Mapping Utilities

There is an ever-growing list of free and open-source software (FOSS) utilities that allow users to define customized network probes and scans [46]. One of the most popular is Nmap (Network Mapper) [47], a utility for network discovery and security auditing. It

is a network mapping utility in that it can perform various methods of network pings and traceroutes to build a topological map of a network, but also performs as a scanning utility in probing port numbers on hosts to assess the type and version of services running, and to fingerprint the operating system.

2.3 Mapping Deception

Generating accurate topological network maps can be a complex process, with many different methods of measuring employed, each with an array of different tools, all in attempts to determine ground truth, though we continue to get different results [48]. The design and complexity of the Internet alone can lead to false or ambiguous representations of a network route. As well, there is an assortment of network technologies whose sole purpose is to provide deceptive representations of network services.

2.3.1 Unintentional Deception

The scale of the Internet has grown to a global network of networks beyond what early developers could possibly have ever imagined. Defined by thousands of RFCs and hundreds of networking protocols and standards, all left to be interpreted by those who develop and implement networking equipment and solutions to ensure interoperability, at best, there are bound to be difficult to interpret probing anomalies. Even the many tools available to perform measurements are imperfect and limited in their ability to accurately represent the true network state at the point of observation.

Previous research [49] compared the results of probes from different vantages points across the Internet, using various utility methods and comparing the results. They found that different probe types, ICMP, UDP and TCP, often presented a different topology representation of the probed network. The anomalies between methods could be the result of certain UDP or TCP port traffic being blocked by some routers, or even the common practice of just blocking ICMP traffic. Failure to get some responses could simply be certain routers configured to not provide error messages for dropped packets, or the router operating at such a high utility that ICMP error messages are not generated. Given that UDP is a non-reliable protocol, it is possible that a UDP probe packet was simply lost along the way to the destination. As well, some routers may put a lower priority on processing ICMP traffic, that the ICMP probe or the response to a probe is dropped by routers operating over partic-

ular thresholds. Since ping and traceroute utilities will only listen for a short duration for a probe reply, it is possible that delay queuing and processing of ICMP messages by routers, could cause delay of a probe reply message such that it arrives after the listening period has expired.

Some of the most common sources of anomalies in network measurement is the use of load-balancing routers and optimal path packet routing. With load-balancing routers, more than one path between sets of routers are used and traffic flows are broken up between these paths. With optimal path packet routing, a response packet could take an alternate route back to the source, as was discussed in §2.1.1. Eliminating the load-balancing anomaly when conducting network traces, is the primary purpose behind the development of Paris-traceroute [45]. The developers of paris-traceroute highlight several methods commonly employed in load-balancing network traffic, each causing different effects with ICMP, UDP and TCP traffic flows, and hence different results with different probing methods. By exploring the various methods in which network load balancing in employed and configured, the developers slightly modified the way in which the probe packet header fields are constructed in paris-traceroute. This change allows all probes towards a target to follow the same path in per-flow load balancing, but still in a fashion to distinguish each probe packet to properly associate its reply message. Though paris-traceroute overcomes anomalies from per-flow load balancing, challenges can still be present with per-packet load balancing.

There is an ever-growing array of network security tools, applications, and appliances designed to protect networks from undesirable traffic. These protection methods can prevent or minimize an adversary's ability to probe and map a networks' topology, and though they are not specifically designed with the purpose of deception, they are worth briefly mentioning here. One of the most basic of network security controls is to filter out undesirable traffic from entering a private network. The most common method to employ traffic filtering is to utilize a feature inherent with most network routers, that of Access Control Lists (ACLs). These are simple rules that packet headers are checked against as they enter a router, where the packet is either allowed or denied through based on the specific ACL rules defined. These rules can filter traffic on source and destination IP address, network protocol, source and destination port number, as well as filter traffic based on already established TCP connections and the direction the packet is traveling through the network. ACLs are

most common method of filtering ICMP as discussed in §2.2.3, and primary employed at the network edge.

Router ACLs do have their limits with traffic filtering as the primary purpose of a router is to move traffic between networks. Firewalls on the other hand are designed specifically for advanced traffic filtering, are either software based for installation on a specific host, or hardware based and installed as a network appliance. Firewalls can control and filter traffic by the same criteria as ACLs, but are capable of preforming much more fine-grained packet inspection and offer much greater logging controls. Both ACLs and Firewalls, when properly employed, can impose challenges to those attempting to probe networks for topology mapping. However, these are primarily static controls and mostly allow or deny traffic based on the pre-configured policy. An IPS can go even further in providing some reactive filtering on a network by monitoring traffic based on a defined policy, and actually control and prevent specific traffic flows to include generation of connection resets to network sessions and probe packets. Again, these security controls are not specifically purposed to intentionally provide deception, they can impact network topology mapping efforts. There, however, other services that are specifically purposed to present deception in a network.

2.3.2 Intentional Deception

There is a category of tools that are designed specifically to present deceptive services on a network. Their purpose is to lure a potential adversary away from primary network resources and learn their Tactics, Techniques, and Proceduress (TTPs). As well, we are exploring additional methods to further present an adversary with a deceptive representation of the network that can build on or even enhance current deceptive technologies.

Honeypots are security tools specifically intended for deception on a network as they represent a false service, computer server, or entire network. They are deployed in live networks to intentionally draw in an adversary away from actual network services, enticing them to probe, scan, and even attempt to exploit the faked services with the purpose of learning their TTPs. Honeypots generally fall into one of two categories, low-interaction or high-interaction. The low-interaction honeypots typically are virtual machines designed to present a small set of basic services that an adversary may seek to exploit, and usually hidden enough that a casual user would not happen upon them. On the other hand, high-

interaction honeypots can scale to representing an entire network with extensive services offered and providing a rich depth of interaction to a potential adversary. Though these services can also be deployed as virtual machines, they may span multiple servers and provide extensive security to ensure the adversary remains fully isolated in the honeypot services. Some of the most prevalent groups conducting work in this area are The Honeynet Project [50] and Project Honey Pot [51], with many different open-source and proprietary solutions that can be implemented in a network to deceive attackers.

Though honeypots are primary focused on presenting false end-point services, none extend to providing deceptive routes to those honeypots, or even deceptive routes to actual network resources. Hence was the focus of some recent research [16] in designing a technique for presenting a deceptive dynamic network topology to anyone attempting to probe a network to construct a topological map of that network. The general research thought was that we can simply filter such adversarial traffic using some of the various network security methods previously discussed. However, the argument was also that given such blocking methods, it was questionable whether we truly prevent the network from being probed by an adversary with blocking means alone. The methodology implemented in this deceptive dynamic network topology, employed a bump-in-the-wire, multi-homed, customized kernel Linux computer that inspected all traffic destined to an internal network server. If that traffic matched packet header fields characteristic of network probing based on the defined policy, a customized response would be sent back to the source of the probe to essentially paint a false topology of the network leading to that host. By properly choosing a deceptive topology to falsely represent to probes in the networks, we can make the network appear weak where it is strong and strong where it is weak. While the concepts are sound and the results were promising in representing a proof-of-concept solution, it is questionable whether such a solution could scale well in an enterprise network environment, and hence the purpose of this continued research in exploring the premises of SDN and the OF protocol in constructing a more agile solution for employing a dynamic deceptive network topology.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: SOFTWARE-DEFINED NETWORKING AND THE OPENFLOW PROTOCOL

The field of SDN and the OF protocol, as briefly presented in §1.1.2, are completely changing the way we can design and manage networks architectures. From an application perspective, the network was simply the plumbing that ensured data and information moved from one point to another, with no real knowledge of how the plumbing performed its functions. SDN and the OF protocol allows network engineers the ability to now innovate in the wiring closet, deploying networks that are specifically built for the information environment they support and allowing those applications to gain intelligence of the underlying network.

In this chapter, we will discuss the origins of the OF protocol, and its relation to SDN. We will then discuss the types and purpose of OF controllers, and review the salient features of the OF standard and provide relevant details on its operation.

3.1 The Origins of OpenFlow

It is difficult to pinpoint the exact origins of the OF protocol, but several projects in succession have paved the way to its initial development. The initial concepts of separately dividing network functions into architectural planes seems to be rooted in a paper that proposed a "clean-slate" approach in redesigning the control and management functions in networking equipment from the ground up. The architectural design was called "4D" [52] based on the idea that four distinct and separate planes existed in networking equipment: decision, dissemination, discovery, and data.

An early project by some of the creators of OF was *The Virtual Network System (VNS)* [53], a hands-on networking environment in which students could generate and work with sensitive raw network traffic in their efforts to learn the many aspects of Internet infrastructure. Though methods existed to facilitate this learning, it was limited in that students could not interface actual equipment with the Internet to work with real-world traffic. Acquir-

ing hands-on learning then required use of limited special-purpose simulation programs with non-standard controls, or extensive networking labs using dedicated hardware, which took many hours to configure. As the name implies, VNS allowed students to easily construct virtual networks in a protected topology and process real Internet traffic. Being a virtualized environment, students could connect remotely from anywhere on the Internet to participate in various projects.

A later project in 2006, *Secure Architecture for the Networked Enterprise (SANE)* [54], of which the creators of VNS were also a part, sought to address the limited flexibility of managing and controlling networks with current complex network appliances and security mechanisms. The idea was to implement a *protection layer* that spanned across all enterprise equipment to control all routing and access decisions throughout the enterprise. This layer would reside between the Ethernet and IP layers of TCP/IP. The enterprise policy would be managed from a logically centralized server to define *capabilities* for communication within the enterprise. SANE's architecture was constructed around the following design goals: allow natural policies that are simple yet powerful, enforcement should be at the link layer, to prevent lower layers from undermining it, hide information about topology and services from those without permission to see them, and have only one trusted component. The initial prototype for SANE was constructed within VNS.

Building on the conceptual successes from SANE, the same creators went on to support a project in 2007 entitled Ethane [55]. Where SANE sought to add an additional layer in the network stack on all hosts and routers to control enterprise communication, it only managed to hide the complexity of the network, vice reduce it. Ethane maintained the concepts of enterprise communications management through a centralized controller, but introduces the use of simplified flow-based Ethernet switches by which the communications policy throughout the enterprise can be managed. It was considered simplified in that an Ethane switch does not need to perform common layer 2 switch functions of learning Media Access Control (MAC) addresses, or supporting Virtual Local Area Networks (VLANs), nor upper layer 3 and 4 functionality either. Three fundamental principles that defined Ethane were: the network should be governed by policies declared over high-level names, the policy should determine the path that packets follow, and the network should enforce a strong binding between a packet and its origin. Ethane was constructed at Stanford University

and connected several hundred registered hosts with several hundred users. As vendor switches mostly allowed only the use of proprietary firmware, the wired Ethane switches were constructed from NetFPGA [56] programmable cards and software switches built upon the Linux operating system. The only modified vendor switches used in Ethane were a common brand name, home office wireless access point, running a FOSS Linux distribution for embedded devices.

Drawing much from previous work conducted with Ethane, OpenFlow [57] was first presented as a white paper in early 2008 as a way for researchers to run experimental protocols in an isolated environment, but using portions of the existing Campus Area Network (CAN). The targeting of campus networks for implementation of OF was part of the continuing push for "clean-slate" network architecture research presented by the 4D paper in 2004. OF enabled this environment to be built upon existing campus network infrastructure already deployed and in use by everyday users. The general idea was to partition network switches and routers into production and research flows by physical network port, allowing the network device to manage flows for production traffic on the campus network assigned ports and OF to program the flow-tables for ports assigned for research. These dedicated OF switches consists of three parts: a flow-table on the switch that contained match criteria for flows and specified an associated action for each flow, a secure channel to interface the switch with a controller, and the OF protocol itself. Therefore, the OF protocol provides the communication between a controller and one or more OF switches for setting flow policy within the network. The initial specification consisted of a 10-tuple for packet header fields by which flows could be distinguished and matches conducted, as further detailed in § 3.4.

Though initially licensed by Stanford University, an OpenFlow Consortium was established with the sole purpose to maintain and grow the OF specification within the research community. The consortium established a public website (www.openflowswitch.org) to publicize the OF standard and foster community growth of the standard; later changed to www.openflow.org [6]. The consortium continued to develop the OF standard while working with several industry switch manufacturers in their creation of OF-enabled vendor switches. Given the growing interest and development in OF from the networking industry, the Open Networking Foundation (ONF) [5] was established in 2011 to promote adoption

of SDN through open standards development. With its establishment, the ONF took over management of the OF standards, transitioning the standard to enable product commercialization. In its first year established, the ONF had over fifty member companies. The foundation now consists of over 100 member companies in early 2014, with thirteen different active working groups, and has released several iterations of the growing OF standard to date.

3.2 Premise of SDN

It is important to distinguish that OF is not synonymous with SDN and even though much work within SDN over the past few years is with the free and open-source OF protocol, using the OF protocol is not required in an SDN architecture. Some major vendors in the networking industry, along with supporting OF in their products, are hard at work in developing their own proprietary SDN solutions [58], [59], some of which can work along side or in place of the OF protocol. Even though SDN would appear to be a new technology, its general concepts of separating layers and gaining more programmable control within a system have been around for much longer as further detailed in a recent paper covering the history of SDN [60].

As mentioned in §1.1.2, and derived from the ONF White Paper on SDN, the primitives of SDN, and specifically the OF protocol, are:

1. Decoupling of the network switch management control plane from the data plane.
2. Centralized controller to manage one or more switches.
3. Ability to manage the behavior of the network using well-defined interfaces and standards based programs.
4. The ability to manage devices across multiple vendor switches through non-proprietary programmability.

The basic SDN architecture in Figure 3.1 [8], is logically represented as three layers: Infrastructure Layer, Control Layer, and Application Layer. Common discussion in the industry regarding SDN networks is normally from the perspective of the Control Layer where the SDN controller resides, as the controller is the central component that manages an SDN network. Anything in the Application Layer is usually referred to as northbound, as it resides above the Control Layer; and the Infrastructure Layer is commonly referred to as

southbound, as it resides below the Control Layer. The underlying Infrastructure Layer consists of network devices (from one or more vendors), communicating with one or more SDN controllers at the Control Layer. OF is the controlling protocol providing a standard interface from the SDN controller, pushing the network policy southbound to each network device (again, to one or more vendor switches), effectively decoupling the control plane from the data plane. Though the initial concept and this architecture shows a centralized controller, more advanced schemes call for multiple or distributed controllers within an SDN architecture, but that discussion is beyond the scope of this thesis. In some implementations, specific purpose applications are written and integrated within the SDN controller, but the long-term goal with SDN is to clearly define boundaries between the controller and the applications they support through clearly defined northbound Application Programming Interfaces (APIs).

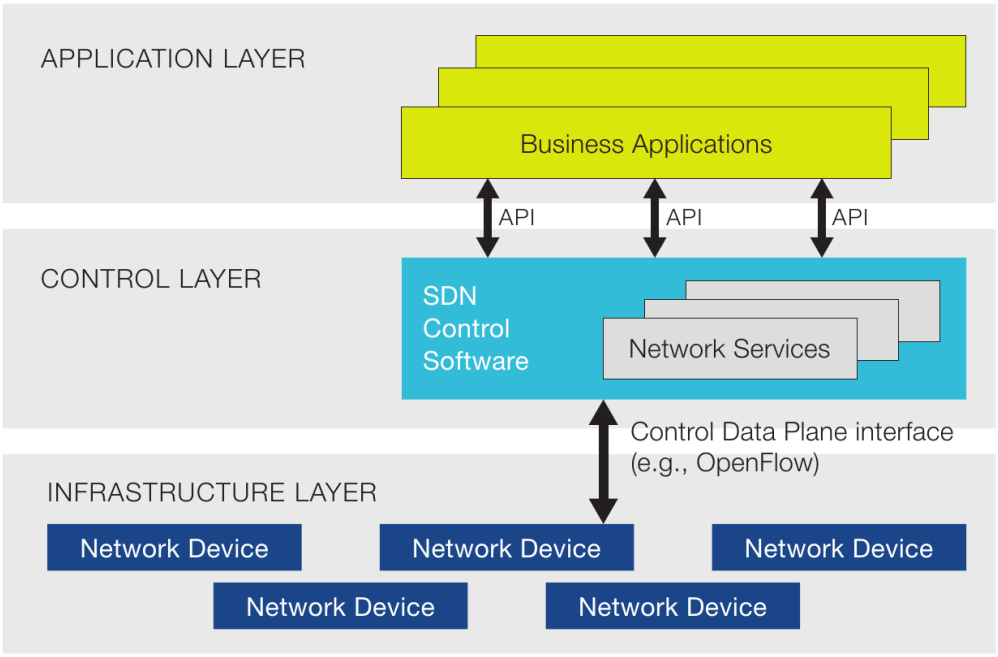

Figure 3.1: Software-Defined Network Architecture. From [8]

3.3 SDN OpenFlow Controllers

The central policy orchestrator of an SDN network is the controller. Ethane did not draw a distinction between the central controller and the communication protocol in use between the controller and the Ethane switches. With the release of the OF protocol, this distinction is established by the simultaneous release of NOX, the first SDN controller that utilizes the OF protocol. NOX is considered an operating system for networks, providing a programming interface for northbound applications to interface with the network, and a southbound interface to manage network devices through the OF protocol. NOX was designed and released in conjunction with the first OF switch specification, and made available for free download [61], [62].

The OF switch specification is not a packaged library for download, but merely defined in the standards document with C style code. SDN controller developers and OF-enabled switch vendors must correctly interpret the standard and code in OF functionality. The controllers are typically built from table top and rack-mounted enterprise computers running some version of the Linux operating system and loaded with additional development tools to facilitate programming the controller's functionality. Specific SDN controller packages can be obtained from their respective website for download and installation. Development of vendor specific and open-source SDN controllers has expanded since NOX was released

Vendor Controllers	**OpenSource Controllers** (programming language)
Big Switch	NOX, MUL, ovs-controller (C)
NEC ProgrammableFlow	POX, Ryu (Python)
Juniper JunOS Space SDK	Trema (C, Ruby)
Cisco onePK	Beacon, Floodlight, IRIS, Jaxon, Maestro, OpenDaylight (Java)
HP VAN SDN	NodeFlow (JavaScript)

Table 3.1: Some common vendor and opensource SDN controllers.

nearly six years ago, as detailed in Table 3.1. It is important to understand that no two SDN controllers are the same, not only with the degree of openness in the source code or the programming language it was developed under, but also the OF switch specification version each supports, which we will further discuss in §3.4. As well, some open-source controller projects are sponsored by major switch vendors, (e.g., Floodlight) [63]. Other controller projects remain strictly community-driven projects, such as NOX and the Python OpenFlow Controller (POX) [61].

3.4 The OpenFlow Switch Specification

Even though the OF protocol debuted in 2008 with the initial version of the OF switch specification, the first major version actually considered ready for industry adoption and implementation was not until version 1.0.0, released December 31, 2009 [64]. There are several different iterations of OF switch specification documents released since then, each providing additional functionality to the OF protocol and available on ONF's website [5]; with major changes between iterations detailed in Appendix A. As previously mentioned, no actual OF library exists to download and plug-in to an SDN controller or vendor switch to implement a particular OF switch specification version. The creators of each SDN controller and OF switch must correctly interpret the specifications documents and implement the version standard correctly, hence why subsequent specification documents include a long list of changes and clarifications to minimize ambiguity.

Since OF version 1.0.0 was the first that vendor support was expected, and by which any equipment claiming to be OF-enabled must at a minimum support, our discussion will primary focus on functionality from this version. An OF-enabled switch is one for which the vendor has coded the switch firmware to facilitate communication with the OF protocol, applying the policies and rule-set pushed to it from an SDN controller. Initial specifications by default established communications over TCP port 6633, however, use of that port was never registered with the Internet Assigned Numbers Authority (IANA) and subsequently assigned by IANA for other services. OF switch specification version 1.0.2 [65] and 1.3.3 [66] highlight the change to IANA assigned TCP port 6653 for switch-to-controller communication.

An OF switch normally initiates a connection to an SDN controller that it has been pre-configured to communicate with, though recently released OF switch specification version 1.3.2 [67], if implemented by the SDN controller and OF switch vendor used in a network, provides a method for the controller to instead initiate the connection with the switch. Through this switch-to-controller session, various control and management message types are used to enable the controller to poll information about the switch, for the switch and controller to manage traffic based on flow entries established by the set policy, and simple session management messages to keep the connection alive. If the switch loses communication with a controller, it can be configured to operate in one of two modes: fail-secure, where traffic continues to flow based on table entries until those entries expire; or fail-standalone, where the switch expires all table entries and operates based on the pre-configured default switch behavior.

A basic OF switch consists of a single flow-table by which the controller can set flow entries on the switch based on packet header match fields, defined by the configured policy pushed to it by the SDN controller, as illustrated in Table 3.2. Each entry in the flow-table has associated counters to track the number of matches against each flow entry. All packets that enter the switch are processed against the table entries, much like a packet entering a router can be matched against an ACL. A match against an entry will cause the packet to be handled based on the flow entry's set action. A failure to match a packet against an entry in the flow-table, known as a table miss, will by default forward the packet to the SDN controller for processing. As well, we can specifically set flow match rules with the specific action to forward a packet to the controller. This action and more will be discussed in more detail below and in §3.4. Packets can also be injected from the SDN controller through a connected OF switch and sent out on the network.

Header Fields	**Counters**	**Actions**

Table 3.2: A flow entry consists of header fields, counters, and actions. From [64]

An OF switch is limited, however, in the number of header fields by which it can match packets as illustrated in Table 3.3. By comparison to the OSI model layers, we match fields in layers one through four. At layer 1, the physical layer, we can match a packet based on the physical port it arrived. At layer 2, the data link layer, we can match a packet on the Ethernet source and destination address, and the Ethernet type. If VLANs are used in

the network, we can also match based on the VLAN Identification number and the VLAN priority [68]. At layer 3, the network layer, we can match a packet based on the IP source address and destination address, the IP protocol, and the IP Type-of-Service (ToS) field. Most traffic across a network will typically be TCP or UDP as discussed in §2.2.2, and as such, we can match at layer 4, the transport layer, the source port and destination port. However, we also discussed ICMP in §2.2.1, a common protocol for error reporting between host. A match of the IP protocol to be ICMP, will instead of source and destination port, use those match fields as ICMP Type and Code. When creating OF match rules for the header fields, we are limited to specifying either a specific value or the default "ANY." The one exception to the match field value limitation is the addition of maskable datalink and network source and destination addresses added in OF switch specification version 1.1 [69], where we can test for a range of MAC and IP addresses. Otherwise there is no way to specify a range of values to match in a particular field, such as if we wanted to match a range of TCP or UDP port numbers.

Ingress Port	**Ether Src**	**Ether Dst**	**Ether Type**	**VLAN ID**	**VLAN Priority**	**IP Src**	**IP Dst**	**IP Proto**	**IP ToS**	**Src Port**	**Dst Port**

Table 3.3: 12-tuple fields to match packets against flow entries. From [64]

A break down of the Ethernet frame is illustrated in Figure 3.2, showing the potential OF match fields highlighted. Reading the frame from left to right, there is nothing to be gained by matching against the preamble or Start Frame Delimiter (SFD) as these are common amongst all frames in an Ethernet network. The Preamble is a special 7 byte series of alternating 1's and 0's to allow a switch to synchronize to an incoming frame and the SFD

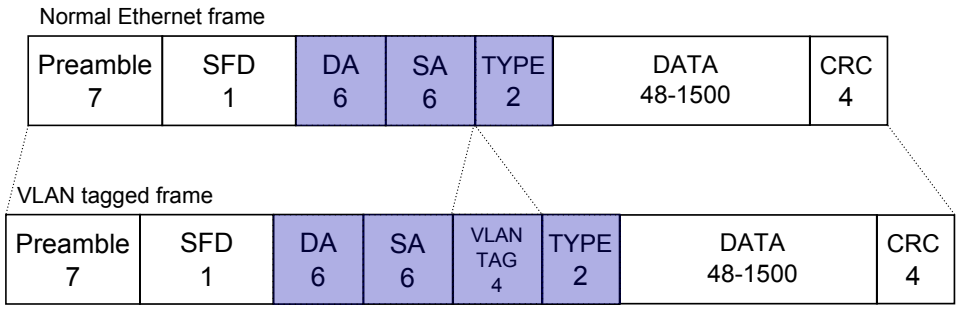

Figure 3.2: The Ethernet header with OF matchable fields highlighted.

is a special 1 byte (01111110) to indicate to the switch that what follows is the actual frame header to be read. The first useful information in an Ethernet frame is the Destination Address (DA) and the Source Address (SA), which are a 6 byte (48 bit) MAC address to uniquely identify physical nodes on an Ethernet network. If VLAN tagging is used in the network, OF can match specifically on the VLAN Identification number (12 bits) and the VLAN User Priority (3 bits). The TYPE field in an Ethernet frame is to indicate the type of payload carried in the data portion of the frame: 0x0800 for IP, for 0x0806 for Address Resolution Protocol (ARP), 0x8100 for VLAN, etc. The data portion of a frame is simply the frame's payload, layers 3 through 7 of the OSI layer with their individual headers and payloads. Lastly, the Cyclic Redundancy Check (CRC) is a method to determine if an error occurred in the transmission of the frame, if the CRC check fails, the frame is simply dropped.

The primary purpose of a router is to move packets between networks, and the IP header information, as illustrated in Figure 3.3 with the potential OF match fields highlighted, is the main criteria by which routers make their decision. The first field of interest is the IP version: IPv4 or Internet Protocol version 6 (IPv6). As we are focused on discussion of OF switch specification 1.0.0, this field is not highlighted, however, support for IPv6 was added in late 2011 with the release of OF version 1.2 [70]. The first highlighted OF matchable

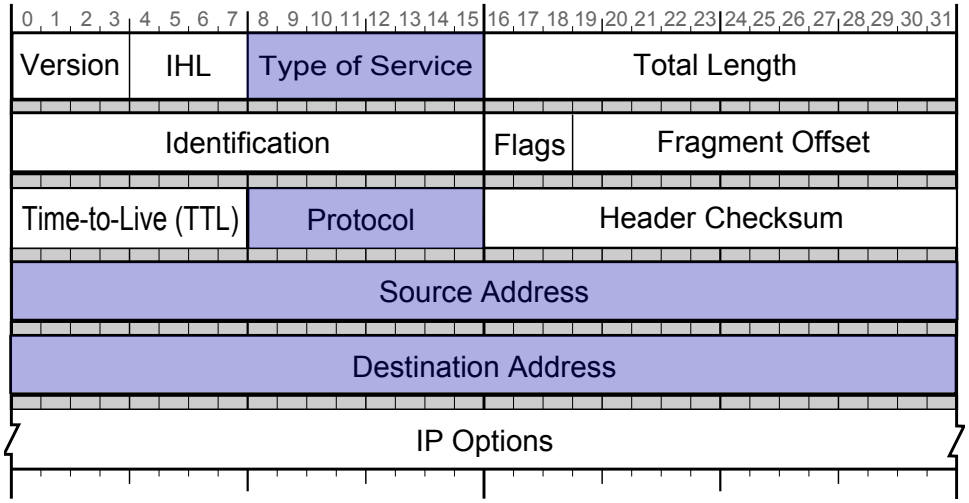

Figure 3.3: The IP header with OF matchable fields highlighted.

field of interest is the ToS field, which originally under RFC 791 [43] allowed assigning a priority to each IP packet and request special handling of the packet. Many representations of the IP header still indicate this field for ToS, and the OF switch specification code it as the "NW_TOS" field. However, RFC 2474 [71] redefined the first 6 bits this field for Differentiated Services Code Point (DSCP) to provide more detailed marking of packets for Quality of Service (QoS) controls; and RFC 3168 [72] redefined the last 2 bits for Explicit Congestion Notification (ECN) to indicate if congestion was experienced in transit. We can also match on IP protocol, which numbers from zero to 255, and commonly used values are 6 for TCP, 17 for UDP, and 1 for ICMP. Lastly, we can match on the IP source address and destination address. What is interesting to note are some of the other fields that OF cannot match against, which could be useful depending on our implemented purpose. The Flags and Fragment Offset has to do with packet fragmentation, and since we are unable to match against these fields in the current OF standard, we are unable to do anything special with fragmented traffic at the switch level. Another interesting field that would be useful to match against is the TTL value, as previously discussed in §2.2.4. Lastly, it would be interesting to dig into the optional "IP Options" portion of an IP packet, but the potential variance in the way options are constructed could be quite complex to implement.

Finally, we can perform OF match functions against the source and destination port number in a UDP and TCP header, as highlighted in Figure 3.4 and Figure 3.5, respectfully. There is no more to be gained from a UDP header beyond the port numbers, as the length only identifies the size of the UDP payload and the checksum is designed to detect errors. There is, however, a wealth of potentially useful information that could be gained through matches in a TCP header, but current OF standards do not enable matches against these fields. For example, having view into the Sequence Number, Acknowledgment Number, and the many TCP flags could allow a detailed stateful inspection of traffic, but this would also

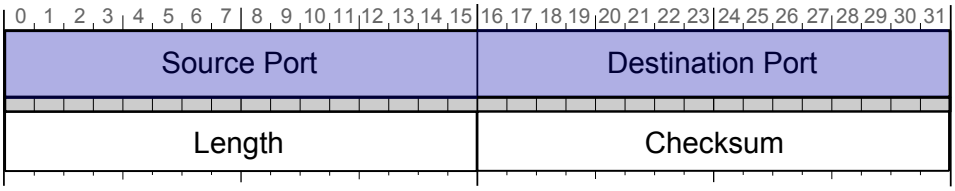

Figure 3.4: The UDP header with OF matchable fields highlighted.

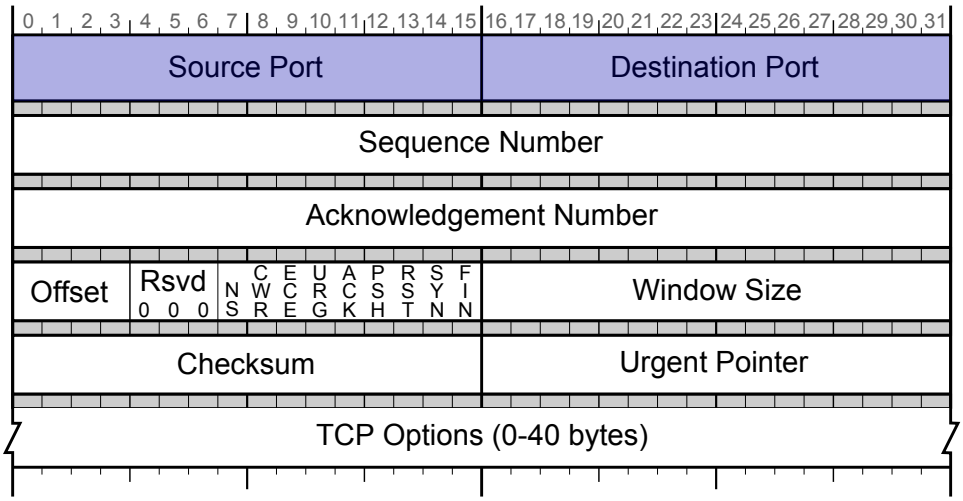

Figure 3.5: The TCP header with OF matchable fields highlighted.

require significant memory capabilities of the switch to maintain that state information. Last, TCP's optional "TCP Options" would be interesting to match against, but as with the optional IP options, the potential variance in the way the options are constructed could be quite complex to implement.

As we have explained, there are many ways we can construct match criteria, though limited, for flows in a OF switch table. Having access to the full range of header information to match packets with OF could allow creation of extensive SDN solutions with OF-enabled switches. There are other tools, such as netfilter's iptables [73], that can be installed and configured on a basic Linux host, which has extensive kernel level hooks to the network stack, and allows full range management and manipulation of packets. As such, this allows employing iptables for stateful and stateless firewalls, proxies, IDSs and IPSs, and many other solutions. Unless the OF standards were further extended to include these additional packet header fields, thereby achieving the same granular packet control and manipulation as iptables, we instead have to perform less granular flow matches on packets and forward those packets to the controller, where we could then employ other tools on packets at the user level. We will further discuss the impact of these limited OF match fields in supporting our deceptive dynamic network topology in follow-on chapters.

There are some considerations when constructing match conditions for flows, in that matching at a higher layer, requires at least one supported match other than "ANY" at a lower layer. For example, to match VLAN ID and Priority, there must be an Ethernet Type match of 0x8100 for VLAN. Setting a rule with Ethernet Type of "ANY" will not enable checking for specific VLAN tagging information within a frame. If we want to match network layer header information such as source and destination IP addresses, then we must at a minimum match Ethernet Type at the datalink layer of 0x0800 for IP. To move up the stack and match UDP or TCP port numbers, then we must match the associated protocol number in the IP header, and have matched Ethernet Type at the datalink layer of 0x0800 for IP. Matching ICMP Code and Type is similar in that we must match the IP protocol as ICMP, and have matched Ethernet Type at the datalink layer of 0x0800 for IP. Though this would seem to indicate the ability to inspect for malformed and corrupted packets, OF Switch Errata version 1.0.1 [74] provided clarification that "the specification does not define the expected behavior when a switch receives a malformed or corrupted packet."

As previously mentioned, each of these table flows will have associated counters maintained on the OF-enabled switch and methods by which the controller can query these

Action	**Description**
Required:	
ALL	Send the packet out all interfaces, not including the incoming interface.
CONTROLLER	Encapsulate and send the packet to the controller.
LOCAL	Send the packet to the switch's local networking stack.
TABLE	Perform actions in flow-table. Only for packet-out messages.
IN_PORT	Send the packet out the input port.
Optional:	
NORMAL	Process the packet using the traditional forwarding path supported by the switch.
FLOOD	Flood the packet along the minimum spanning tree, not including the incoming interface.

Table 3.4: Required and optional OF table match actions. From [64]

counters. But these flows are incomplete without also defining an associated "Action" for a match against a flow entry. We previously mentioned one default action for a packet, which is to forward the packet to the controller on a table miss, where there is no matching flow entry on the switch. The other default action for a packet is when a specific flow entry simply does not specify an action, in which case the default action then is to drop the packet. Table 3.4 provides the required actions that an OF-enabled switch must support for physical and virtual ports, and some additional optional actions that the switch can support for virtual ports. Again, we will make specific use of the "Controller" action in matching probe packets to forward to the controller so as to craft customized deceptive response packets to present a false network topology.

When an OF switch starts up and establishes a connection to the controller, the flow-table of that switch will be empty, unless some pre-configured flow entries are pushed to the switch. Recall that the default action on a table miss, a packet fails to match any flow entry in the flow-table, is to forward the packet to the controller for processing. Therefore, the controller will normally receive an initial surge of packets forwarded to it as table misses occur on the switch, because of these lack of flow-table entries. Based on the configured policy on the SDN controller, flow entries will then be generated from the controller-received packet and pushed to the switch for inclusion in the flow-table to match future packets. As packets continue to arrive on the switch, they will be checked, line-by-line, against the flow-table entries on the switch for a match. If a match is made on a flow entry, the pre-defined action for that flow as listed in Table 3.4 is performed on the packet, and no additional entries further down the flow-table are checked. This could present a problem as some flow entries may be more granular than others, potentially allowing certain packets to match on a flow entry and an action taken that was not intended.

As an example, say we want to block all ICMP packets, but allow all other traffic, regardless of protocol or port number. The easiest way to simply allow all other traffic through is by just matching on Ethernet Type of IP. However, as soon as a single packet that is not ICMP arrives at the switch after startup, is then sent to the controller due to a table miss and processed by the controller, and a flow entry created on the switch, the lack of granularity in the flow entry would then also match and allow all ICMP packets through the network instead of blocking them. This is because ICMP is carried on IP and the rule established

allowed all IP through, including the ICMP packet. To alleviate this problem, we could simply establish more granular match criteria for the non-ICMP traffic as well, such as a separate rule for TCP and UDP traffic, but now we also have to specify how the controller is to handle other protocols as well. A simpler method to manage the ordering of multi-field flow entries on the table is to set the priority level for the flow entries pushed to the switch, which is a 2^{16} value field, allowing a priority range from 1 to 65535. The higher the number, the higher the priority of the flow entry in the table. As such, more restrictive flow rules should be set with a higher priority to be checked first, and in our example, we would set the ICMP flow entry with a higher priority number than the less granular flow entries. This would ensure an incoming packet is first checked against the more restrictive rule to block ICMP.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: METHODOLOGY

In this chapter, we present the specifics of our constructed test environment and the details of how we implemented our deceptive topology through SDN and the OF protocol. We first discuss the design overview of our experiment and the reason for an initial limited approach to blocking specific types of probe traffic. We then discuss the network design to include routers, switches, etc., along with the choice of software selection in our test environment. Next, we discuss the details of the SDN controller and specifically how the OF switch is configured. Last, we describe the details of network operations to include switch-to-controller communication establishment, to deception policy configuration, and how probe packets are handled and the deceptive responses generated.

4.1 Design Overview

As discussed in Chapter 2 traceroute probes can be constructed using ICMP, UDP, or TCP packets. Previous work, [16], focused on detecting and responding specifically to UDP probes, however, as ICMP is primarily used for network testing and error reporting, we will focus our efforts first on detecting and responding to ICMP traceroute probes. As well, ICMP is more simplistic in that it typically does not carry application level data as a UDP or TCP packet is designed to do. Therefore, we need not focus on differentiating between legitimate data communications and network probes with ICMP as we would with UDP and TCP packets. We could specifically define rule-sets for the full range of ICMP packet types, however, for simplicity sake, we will only focus on matching ICMP Echo messages and provide equivalent ICMP Time Exceeded and Echo Reply messages as discussed in §2.2.1. More importantly, ICMP Echo messages are really the only packets that are likely to get through anyway, and are what is typically used when doing ICMP-based traceroute. Therefore, all other ICMP Types and Code messages will be dropped as no rule is defined to process them.

As discussed in §3.4, once a packet is sent to user space on the controller, we can employ other methods to fully manipulate individual packets. From the controller we can inspect the IP TTL and other packet header information to craft our customized reply messages to

individual probes. These reply messages can then be sent to the switch to be forwarded to the probe originator.

4.2 Network Design

Since the focus of our research is to determine if the premises of SDN and the OF protocol can enable employing a dynamic deceptive network topology as initially presented in previous work [16], we modeled our network design to closely mimic the design employed in that work; as depicted in Figure 4.1. Our experiment also used a simple three-node network comprised of the adversary's workstation and a target web server, but instead of a custom designed *intelligent router* to separate the two, our SDN solution has in place an OF-enabled switch connected to an SDN controller. The only actual physical node in our experiment network is the OF-enabled switch, interfaced to a multi-homed desktop computer that hosts the rest of the test network. All other nodes are constructed in a virtual environment running on a multi-port desktop computer. The interface links are all Ethernet, running Full-duplex at 100 Mbps. With the exception of the switch-to-controller link (in blue), each network segment was configured with a standard class C network block of 172.20.x.x/24. The link connecting the switch and controller was configured with a standard class C network block of 192.168.x.x/24. Detailed interface addressing is indicated

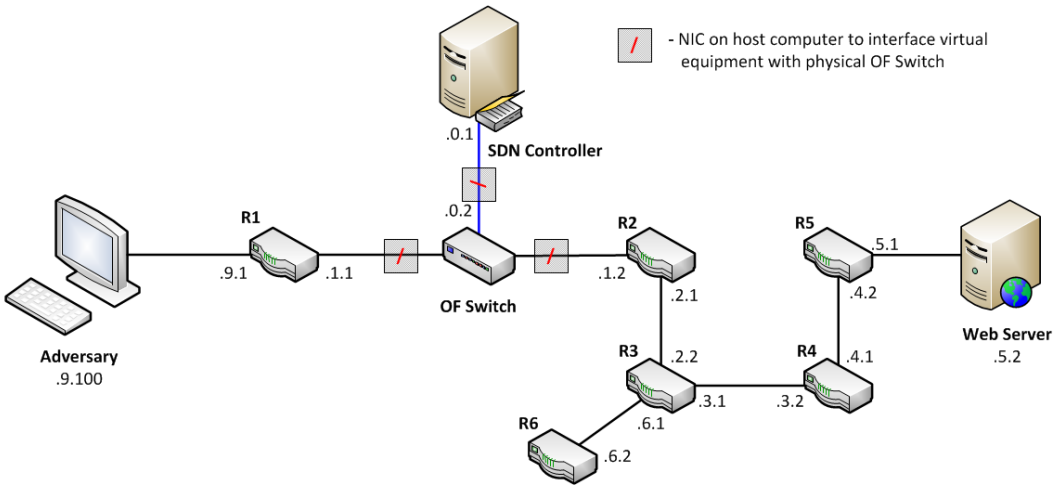

Figure 4.1: Experiment topology implemented in virtual environment. All IP addresses are on the 172.20.x.x network, with the exception of the SDN controller interface to the OF switch on the 192.168.x.x network.

in Figure 4.1. The detailed physical port interfacing of the OF switch will be discussed further in §4.5.

Given that the goal is to prevent an adversary from accurately mapping the true topology of the protected network, we deploy our deceptive topology generating solution near the ingress point of the private network. In this case, R1 is the border router separating the private network link from the service provider link. Any host probing the private network would observe ground truth of the network topology up to the border router (R1), afterwards all other responses to network probes would be provided by the deceptive solution. The OF switch itself is not visible to network probes since it is not acting as layer 3 router.

4.3 Software Selection

The host for our experiment network was built on a single desktop computer running the Ubuntu Linux 12.04.3 [75] operating system. We used Graphic Network Simulator (GNS3) [76] to design and configure our virtual network environment of routers, adversary host, web server, the SDN controller, and to interconnect the physical OF network switch. This allows us the flexibility to build and modify our test architecture without the need to use expensive physical hardware and significant physical space to house the test network. GNS3 makes use of additional emulators to run actual router Internetwork Operating System (IOS) software and to run the guest virtual machine operating system. This enables the routers and guest machines to provide full functionality in the network, equivalent to what a dedicated physical router and physical server would provide.

Each of the routers in the experiment topology (R1 through R6) was a virtual Cisco 3725 router running Cisco IOS Version 12.4(15)T14, release software with field change 2. The choice of router was simply availability of actual router IOS software from on-hand physical Cisco routers. Actual implementation of our SDN solution, with an SDN controller and OF-enabled switch, could be deployed in any network environment with smaller home-office routers or large enterprise grade routers, depending on the needs of the network. All routers were configured to run Routing Information Protocol (RIP) version 2, a simple distance vector routing protocol to control packet routing within the network.

For the guest virtual machine operating system, specifically the SDN Controller and the web server, also ran Ubuntu Linux 12.04.3, as was used on the host computer. We will

discuss further the specific software configuration of the SDN Controller in §4.4. The only additional software loaded on the web server was a FOSS wiki program to provide a functioning web server for testing. The adversary guest virtual machine was loaded with BackTrack Linux 5.0 [77], a Linux-based penetration testing platform that comes preloaded with numerous utilities, including many network probing utilities.

4.4 SDN Controller Configuration

As discussed in §3.3, there are numerous SDN controllers to select from in designing an SDN network architecture. For simplicity, we chose to use the POX [61] SDN OF controller due to our familiarity with the Python [78] programming language. Additionally, the POX library is somewhat simple to work with as it is largely targeted for research and education purposes. The only dependencies for running POX is Python 2.7 or higher. Otherwise, we simply downloaded the latest branch release of POX, version 0.2.0 (carp), from the GitHub [79] repository to the Ubuntu Linux SDN controller virtual machine. A community-driven POX Wiki [80] provides detailed guidance for installing and running POX, as well as describing POX features and interfaces for working with the OF protocol coded into the controller to build custom SDN controller applications.

We discussed at length the many details of the baseline OF switch specification in §3.4, along with some additional changes to the OF protocol included in later switch specification versions. Major changes between switch specification versions is highlighted in Appendix A. The current POX version 0.2.0 (carp) only supports OF switch specification version 1.0.0, with some additional extensions from switch specification version 1.1, though is sufficient to support our basic analysis and testing of the OF protocol in developing our custom SDN controller.

The POX library directory structure is very simply laid out. In the root folder are several "readme" files, along with two key Python script files: pox.py and debug-pox.py. Both of these Python script files are used to invoke POX and can take several arguments as input. The "debug-pox.py" script automatically provides a host of debugging output for development purposes. Some of the additional arguments can include: specifying an alternating listening port for the SDN controller to listen for incoming connection request from an OF switch; specifying output logging methods; specifying the filename of the cus-

tom SDN controller to load and run; and chaining features from other python files, such as flow visualizers. The "pox" subdirectory contains the core of the POX functionality, to include several subdirectories of example Python code for basic SDN controller functionality. Our customized deceptive controller made use of example code from several provided files: pong.py, hub.py, and example.py. Of the many subdirectories contained the pox subdirectory, the "lib" directory and the "openflow" directory contain the majority of additional Python class files we need to create our custom controller. Specifically, the lib directory contains the packet superclass directory, which contains specific Python class files for working with the common supported protocols, such as ICMP, UDP, TCP, etc. We will make use of the methods from these class files when deconstructing and reconstructing individual packets sent to the SDN controller from the OF-enabled switch. The "openflow" directory contains the POX developed class files as interpreted from the OF switch specification. We will provide specific operational details of our customized controller in §4.6.

4.5 OF Switch Configuration

For our test environment, we used a commercial vendor [81], 24-port, OF-enabled, Ethernet enterprise switch. Our configuration initially ran switch firmware version WB.15.13.0000, which supported OF Switch Specification 1.0.0. We later upgraded to the latest switch firmware release, version WB.15.14.0002, that provided some bug fixes among other changes.

Since the switch was the only physical hardware in use other than the multi-port desktop computer that hosted our virtual test environment, all physical interfaces from the switch were interconnected to physical Ethernet Network Interface Cards (NICs) on the desktop computer. This point of demarcation is illustrated in Figure 4.1 by the small gray box with a red slash. Each physical Ethernet NIC in the computer was associated with a single interface on a virtual network device in our GNS3 managed virtual environment. The internal facing port of router (R1) was interconnected with physical Ethernet port 1 on the OF switch; and the external facing port of router (R2) was interconnected with physical Ethernet port 2 on the OF switch. The SDN controller was interconnected with the switch out-of-band management (OOBM) Ethernet port, which operates on the switches' management plane. Older network equipment typically included a console port, which enabled an administrator to establish a serial connection with a computer, and perform command-line

configuration of the device. The serial port was considered a form of OOBM as normal user traffic would not transit the console port. To establish a telnet or secure shell command line or web management interface with the network equipment over IP, required connecting through one of the switches data ports. This method was considered in-band management and required connecting a dedicated computer to one of the ports, or a using a computer on the network that could reach the network equipment via an IP address. Most newer network equipment now include a dedicated OOBM Ethernet port to allow connection of a dedicated computer for management of the devices, or construct a separate management to manage many network device, without that traffic traversing the normal data paths of the network. As traffic is normally not allowed to cross directly from the data plane to the management plane, this adds an additional layer of security for the switch-to-controller connection.

There are several different ways in which the switch could be configured to participate in an SDN environment. The vendor's switch *OpenFlow Administrator's Guide* [82] provides details of these different configuration methods, and provides the steps we used to configure the OF-enabled switch used in our test network. Details of these configurations are captured in Appendix B, the switch startup configuration file.

4.6 Deceptive Network Operations

Once the OF switch is interfaced in the network and the required OF parameters are configured, the switch will continue to function as a normal layer-two switch until a connection is established with the SDN controller. This is because we chose to configure the "connection-interruption-mode," which defines the behavior of the switch when it loses connection with its configured controller, to *fail-standalone* as discussed in §3.4. This choice was simply for ease in conducting comparative network probing with and without the deceptive topology enabled. With the switch OF parameters configured, and the OF mode enabled on the switch, the switch will poll for its configured controller every 60 seconds.

With the OF switch configured and polling for its associated SDN controller, we then need to launch the customized SDN controller to listen and accept the connection request from the switch, and establish policy on the network. This is performed by by invoking POX as discussed in §4.4 with the necessary arguments. In a production environment, the SDN

controller can be configured to automatically execute the POX script on system startup and to restart the service if it were to fail. For our experiment purposes, we chose to manually invoke POX with the following command syntax:

```
./pox.py openflow.of_01 --port=6653 topodeception
```

Here we see the "pox.py" startup script called with some additional arguments. Normally "openflow.of_01" is called automatically when launching the POX script and does not need to be addressed. However, we include it in our command syntax to change the listener port on the SDN controller in accordance with the new IANA registered port as discussed in §3.4, as this change has not yet been incorporated in the current version of POX installed. The "openflow.of_01" component of POX provides all the required methods from the OF standard to enable communication with OF-enabled switches as discussed in §3.4. The last item in the command syntax is reference to our customized Python POX controller filename, "topodeception.py".

Once the script is run, POX will execute our custom controller in the background. The first step for our custom controller is to import all the specified Python libraries needed to run our custom controller, as specified in the topodeception.py script file. Three Global variables are set for common default system TTLs that will be referenced when generating deceptive response packets: Cisco router - 255, Microsoft Operating Systems - 128, and Linux Operating Systems - 64. Next, POX will execute the defined launch operations from our custom controller and open a listener for incoming connection requests from an OF-enabled switch.

```
def launch ():
    # Open listener for switch to connect to controller
    core.openflow.addListenerByName(''ConnectionUp'', _handle_ConnectionUp)
```

With the controller listening on TCP port 6653 at script startup and the OF-enabled switch still polling for its controller, a connection request should be established within 60 seconds. Once the controller receives the connection request from the switch, a session between the switch and controller will be established. The controller will send the switch a *features request* message to learn the specific capabilities of the switch, such as the number and speed of the ports, OF version supported, etc. Once the session is established and the controller

has learned the switch particulars, we would normally open a listener for incoming packets sent by the switch from either by a flow-table miss, or specifically sent through a "CONTROLLER" flow action. However, since we are employing our SDN solution to perform specific actions on probe traffic, we pre-load two flow rules on the switch before opening a listener for incoming packets.

```
def _handle_ConnectionUp(event):
    # Rule to forward all ICMP traffic to the controller
    msg = of.ofp_flow_mod()
    msg.priority = 2  # Higher number, higher priority
    msg.match.dl_type = pkt.ethernet.IP_TYPE
    msg.match.nw_proto = pkt.ipv4.ICMP_PROTOCOL
    msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
    event.connection.send(msg)

    # Rule to process all other traffic as the switch normally would
    msg = of.ofp_flow_mod()
    msg.priority = 1  # Lower number, lower priority
    msg.match.dl_type = pkt.ethernet.IP_TYPE
    msg.actions.append(of.ofp_action_output(port = of.OFPP_NORMAL))
    event.connection.send(msg)

    # Open listener for incoming ICMP packets from switch
    try:
      core.openflow.addListenerByName("PacketIn", _handle_PacketIn)
      log.info("Listener opened for ICMP traffic")
    except IOError:
      log.info("Error opening listener for ICMP traffic")
```

When the controller signals that the connection with the switch is up, that event triggers the pre-loading of our two flow rules. Recall from §3.4 that we can use flow priorities to control the order of flow-table entries on the switch. As well, to make an upper layer header field match, we must first perform a dependent lower layer header field match. The first flow rule is set with a priority of 2 to match the IP protocol of ICMP. The OF switch action for a match on this flow entry is to forward the packet to the controller for processing. The second flow rule is set with a prior of 1 to match all other IP packets, which will include UDP and TCP packets. The OF switch action for a match on this flow entry is to forward the packet for normal processing, using the traditional forwarding path supported by the switch. With both our pre-loaded flow rules set on the switch, we open a listener for incoming packets sent to the controller from the switch.

```
def _handle_PacketIn (event):
    packet = event.parsed
    # Check for Protocol Type 1 "ICMP"
    if packet.find("icmp"):
        # Reply to probes
        if packet.find("icmp").type == pkt.TYPE_ECHO_REQUEST:
            icmp = pkt.icmp()
            ipp = pkt.ipv4()

            # Set variables for calling attributes from packet
            p_in = packet.find("ipv4")
            pp_in = packet.find("ipv4").payload
            echop = packet.find("echo")

            # Filler for Type 3 and 11 replies message
            icmp_unused = 0 # used as 2-byte in icmp.payload
            icmp_next_mtu = 0 # used as 2-byte in icmp.payload

            # Check if TTL == 1
            if packet.find("ipv4").ttl == 1:
                # perform additional actions

            # Else TTL is greater than 1, Echo Reply
            elif packet.find("ipv4").ttl > 1:
                # perform additional actions

        # Else ICMP was other than Echo Request and will be discarded
        else:
            return
```

On the event of a packet arriving at the controller, we inspect the IP protocol to verify that it is an ICMP packet. With the current configuration, we should not expect any packets other than ICMP to be sent to the controller. However, a potential exists for malformed packets or non-IP packets to arrive at the switch, causing a flow-table miss, and the packet to be forwarded to the controller for processing. We verify that the received packet is an ICMP packet before processing further, otherwise the packet is dropped at the controller. We then inspect the ICMP Type for Type 8, Echo request messages, before processing further. Having verified that the packet for processing is an ICMP Type 8 Echo message, we setup some working variables to simplify constructing response messages. Based on the current depth of our constructed deceptive topology we wish to present, we have only two TTL values to inspect for: TTL equal to 1, and TTL greater than 1. We will address this distinction in TTL shortly. From these two conditional checks, we will further perform

conditional checks on the destination IP address in the probe packet to determine the correct deceptive response.

```
            # Check if TTL == 1
            if packet.find("ipv4").ttl == 1:
                # If DST IP == the near-side router IP ADDR, provide ECHO Reply
                if packet.find("ipv4").dstip == IPAddr("172.20.1.2"):
                    icmp.type = pkt.TYPE_ECHO_REPLY
                    icmp.payload = packet.find("icmp").payload
                    ipp.srcip = packet.find("ipv4").dstip
                    reply_ttl = DEFAULT_RTR_TTL
                # If DST IP == the far-side router IP ADDR, provide ECHO Reply
                elif packet.find("ipv4").dstip == IPAddr("172.20.5.1"):
                    icmp.type = pkt.TYPE_ECHO_REPLY
                    icmp.payload = packet.find("icmp").payload
                    ipp.srcip = packet.find("ipv4").dstip
                    reply_ttl = DEFAULT_RTR_TTL
                # If DST IP == the protected host, provide Time Exceeded
                elif packet.find("ipv4").dstip == IPAddr("172.20.5.2"):
                    icmp.type = pkt.TYPE_TIME_EXCEED
                    icmp.code = 0
                    icmp.payload = struct.pack('!HHBBHHHBBHIIBBHHH', icmp_unused,
                                   icmp_next_mtu, (p_in.v << 4) + p_in.hl,
                                   p_in.tos, p_in.iplen, p_in.id,
                                   (p_in.flags << 13) | p_in.frag, p_in.ttl,
                                   p_in.protocol, p_in.csum,
                                   p_in.srcip.toUnsigned(),
                                   p_in.dstip.toUnsigned(),
                                   pp_in.type, pp_in.code, pp_in.csum,
                                   echop.id, echop.seq)
                    ipp.srcip = IPAddr("172.20.1.2")
                    reply_ttl = DEFAULT_RTR_TTL
                # If DST IP == any other IP address, provide DST Unreachable
                else:
                    icmp.type = pkt.TYPE_DEST_UNREACH
                    icmp.code = 1
                    icmp.payload = struct.pack('!HHBBHHHBBHIIBBHHH', icmp_unused,
                                   icmp_next_mtu, (p_in.v << 4) + p_in.hl,
                                   p_in.tos, p_in.iplen, p_in.id,
                                   (p_in.flags << 13) | p_in.frag, p_in.ttl,
                                   p_in.protocol, p_in.csum,
                                   p_in.srcip.toUnsigned(),
                                   p_in.dstip.toUnsigned(),
                                   pp_in.type, pp_in.code, pp_in.csum,
                                   echop.id, echop.seq)
                    ipp.srcip = IPAddr("172.20.1.2")
                    reply_ttl = DEFAULT_RTR_TTL
```

As discussed in §4.2, our OF switch itself is not visible to network probes since it is essentially a multi-layer switch, capable of reading frames up to layer 3 and layer 4 of the OSI model. Recall from Figure 4.1, the OF switch sits in the link between router (R1) and (R2). Therefore, the OF switch does not participate as a standard hop in the path before the deceptive topology is presented, as did the *intelligent router* in previous work [16]. The switch does allow RIP updates to pass between router (R1) and (R2) using the switch's traditional path. As such, we implemented our deceptive path slightly different as an adversary could learn, through network probing, one of the two ground truth router IP address for the link between (R1) and (R2); specifically that the private network facing interface on router (R1) is configured with IP address 172.20.1.2. We therefore virtually collapse router (R2) and (R5) in our deceptive topology implementation as represented in the previous block of code.

For the first conditional check of a packets with a TTL equal to 1, we are only presenting deceptive responses to probes destined to three IP address. Recall that the default behavior of a network router is to inspect the TTL of an incoming packet, and if the packet TTL is equal to 1 and not destined for a port on that router, to decrement the TTL and drop the packet, sending an ICMP Type 8 Code 0, Time Exceeded in Transit message back to the source of the packet. The first node we present in the deceptive topology is our collapsed router (R2/R5), with the near side interface of 172.20.1.2, and the far side interface of 172.20.5.1. A probe packet destined to either of these two interfaces with a TTL equal to 1 would solicit a normal Echo reply. Since this is the first layer router in our presented deceptive topology, we just set the reply TTL equal to a default router TTL of 255. The next node we present is the web server at address 172.20.5.2, and if a probe packet with a TTL of 1 arrived at the router destined to the web server, the TTL would be decremented to zero and a Time Exceeded packet would be sent back to the host. Since the current version POX library did not fully implement the Time Exceeded type in the *icmp.py* class file, we implemented our own struct to create the reply message payload. For all other destination IP addresses, we could take one of two approaches. We could simply not provide response messages and possibly lead the adversary to believe that some form of packet filtering was in use, or we could simply reply with an ICMP Type 3 Code 1, Destination Unreachable, Host Unreachable, message as the last "else" statement provides. Next we review the actions for a probe packet with a TTL greater than 1.

```
# Else TTL is greater than 1, Echo Reply
elif packet.find("ipv4").ttl > 1:
    icmp.type = pkt.TYPE_ECHO_REPLY
    icmp.payload = packet.find("icmp").payload
    ipp.srcip = packet.find("ipv4").dstip

    if packet.find("ipv4").dstip == IPAddr("172.20.1.2"):
        reply_ttl = DEFAULT_RTR_TTL
    elif packet.find("ipv4").dstip == IPAddr("172.20.5.1"):
        reply_ttl = DEFAULT_RTR_TTL
    elif packet.find("ipv4").dstip == IPAddr("172.20.5.2"):
        reply_ttl = DEFAULT_NIX_TTL - 1
    else:
        icmp.type = pkt.TYPE_DEST_UNREACH
        icmp.code = 1
        icmp.payload = struct.pack('!HHBBHHHBBHIIBBHHH', icmp_unused,
                        icmp_next_mtu, (p_in.v << 4) + p_in.hl,
                        p_in.tos, p_in.iplen, p_in.id,
                        p_in.flags << 13) | p_in.frag, p_in.ttl,
                        p_in.protocol, p_in.csum,
                        p_in.srcip.toUnsigned(),
                        p_in.dstip.toUnsigned(),
                        pp_in.type, pp_in.code, pp_in.csum,
                        echop.id, echop.seq)
        ipp.srcip = IPAddr("172.20.1.2")
        reply_ttl = DEFAULT_RTR_TTL
```

Our next set of conditional checks for TTL value is fairly simple from this point as we are only presenting one layer of depth for routers in our deception scheme. However, depending on the depth and breadth of the deceptive topology we want to present, our conditional checks could become much more complex. Therefore, with only one layer of deception, we can treat all TTL values greater than 1 the same. Any packet destined to either interface of our collapsed router (R2/R5), would receive a standard echo reply. For any packet destined to our web server with a TTL greater than 1, we construct an echo reply with a default Linux TTL, less the number of deception layers we are presenting. In this case, our web server is running the Linux Operating System, which typically uses a default TTL of 64, and since we are presenting one layer of deception, we decrement the TTL of the echo reply by 1. Again, for probes to all other destination IP address not in the network or deception scheme, we simply reply with an ICMP Type 3 Code 1, Destination Unreachable, Host Unreachable, message as the last “else” statement provides.

```
        # Construct the IP Packet for Reply message header
        ipp.tos = 0xc0
        ipp.ttl = reply_ttl
        ipp.protocol = ipp.ICMP_PROTOCOL
        ipp.dstip = packet.find("ipv4").srcip

        # Construct the Ethernet Frame for Reply message header
        e = pkt.ethernet()
        e.src = packet.dst
        e.dst = packet.src
        e.type = e.IP_TYPE

        # Link up parts of Reply
        ipp.payload = icmp
        e.payload = ipp

        # Send Reply back to the input port
        msg = of.ofp_packet_out()
        msg.actions.append(of.ofp_action_output(port = of.OFPP_IN_PORT))
        msg.data = e.pack()
        msg.in_port = event.port
        event.connection.send(msg)
```

After processing the ICMP Echo packet to create an associated reply consistent with the deception we want to present, we need to construct the rest of the packet headers for the ICMP probe response message. The two key IP header fields that we modify to provide deception in the ICMP response packet is the source IP address and the TTL. Recall that we conditionally set these values as "ipp.srcip" and "reply_ttl". We set the response packet TTL, "ipp.ttl", equal to the conditionally set "reply_ttl" value. In addition, we ensure the IP header indicates that the payload is an ICMP message, and set the destination IP address of the reply message equal to the source IP address from which we received the original probe message. One final IP header field value we need to set, to ensure the IP header of a deceptive ICMP response message is indistinguishable from a normally generated message, is the *Type of Service* field. As we discussed in §3.4, this field has been re-designated for DSCP, though is still commonly referred to as ToS. An adversary using various probe utilities may not immediately recognize the difference unless they were doing a deep inspection of all probe response packets, but a normal host or router sets the ToS field value as hex `0xC0` for ICMP error messages.

Finally, we need to construct the Ethernet frame header for the message to be forwarded back to the switch for processing. We swap the Ethernet source and destination MAC address, which addresses the Ethernet frame back to the internal port of router (R1). Next we link up the different layers of the reply message to be packaged into the Data portion of the Ethernet frame. Then we send the frame back to the OF switch to be transmitted out the port that the associated probe message was received on. This action is indicated by the "of.OFPP_IN_PORT" as discussed in §3.4 and specifically listed in Table 3.4.

Even though the deception scheme we employed was relatively simple, there is a fair amount of complexity in processing each probe packet and conditionally providing an associated response. Our Python code is probably not as efficient as it could be to optimize the conditional checks on each packet, and we hard coded many variables and IP addresses to build an initial proof-of-concept. We only presented one layer of deception, with two routers virtually collapsed to represent one, and a single host in the protection scheme. Adding additional layers of depth with more routers, and additional breadth by employing redundant links or even load balancing between routers, would increase the complexity of our deception scheme and the topodeception.py script. Further, we have only provided deceptive responses to the simplest of three types of probe packets, ICMP. Including flow rules and conditional checks to capture and provide deceptive responses to UDP and TCP probes as well, would require significantly more work. We discuss all these challenges and limitations of our customized SDN solution in Chapter 6.

CHAPTER 5:
RESULTS

Our research goal was to determine if the premises of SDN and the OF protocol enables employing topology deception for network defense. In Chapter 2, we discussed several broad network topology measurement efforts with a focus to research, as well some of those with a more nefarious end-goal of network exploitation. We also discussed the underlying network protocols and several tools that exploit those protocols to perform network mapping. In Chapter 3, we discussed the premises of SDN and provided many of the salient features of the OF protocol and its operation. We then applied that information in Chapter 4 in implementing our custom SDN controller to reproduce the deceptive topology generated in previous work [16].

In this chapter, we will utilize several of the mapping tools discussed in §2.2 to test the results of our deceptive topology implementation. We will first provide some general observations of behavior consistent across all our tests and provide some analysis as to the root cause of these observations. Then, we will demonstrate the normal and deceptive results from several ping utilities that probe for the presence of an end host. Last, we will demonstrate the normal and deceptive mapping results generated from two common traceroute utilities that infer the hops towards the target.

Recall from §4.1, our deception scheme focuses on providing deception to ICMP based probes, even though many tools can also construct non-ICMP probes. Therefore, all of our test probes in §5.2 and §5.3 use ICMP and will be initiated from the Adversary computer in our test network as previously discussed in Chapter 4. Figure 5.1 specifically illustrates the test network, showing the normal route that would be discovered without any form of deception implemented, nor any network security controls implemented that would block ICMP traffic. Figure 5.2 illustrates the inferred route that will be presented to an adversarial scan with our custom SDN controller. For the deception scheme, recall that we collapsed router (R1) and router (R5) to present a more believable hop-to-hop route based on the addressing scheme in use. For the ping tests in §5.2 and tracroute tests in §5.3, the probe replies in the "normal" test are from the actual nodes in the network (e.g., the host

172.20.5.2 actually provides the response). For the "deceptive" test, with our deceptive scheme employed, the SDN controller actually provides the reply to probes from all nodes along the presented deceptive path as represented in Figure 5.2.

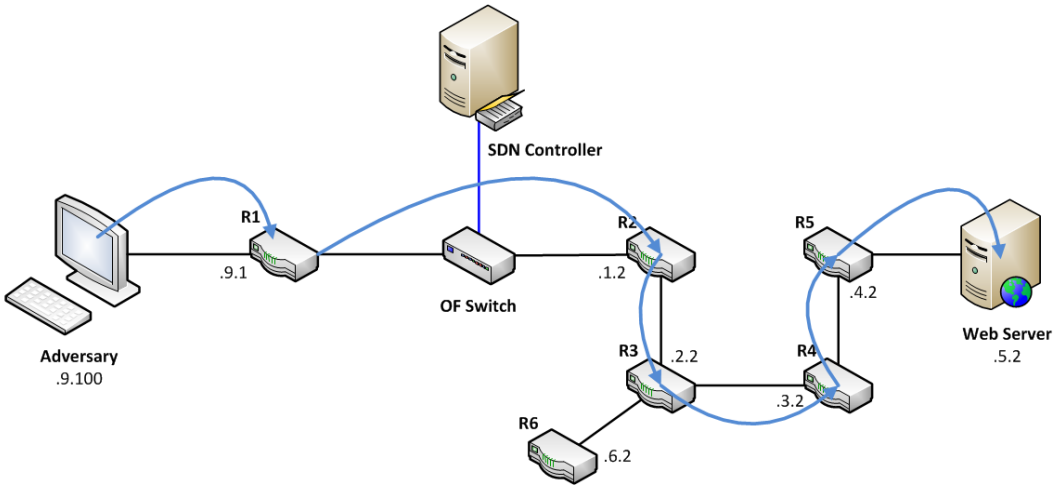

Figure 5.1: The normal route discovered without deception. The solid blue arcs represent the true route provided to the adversary in response to a traceroute probe.

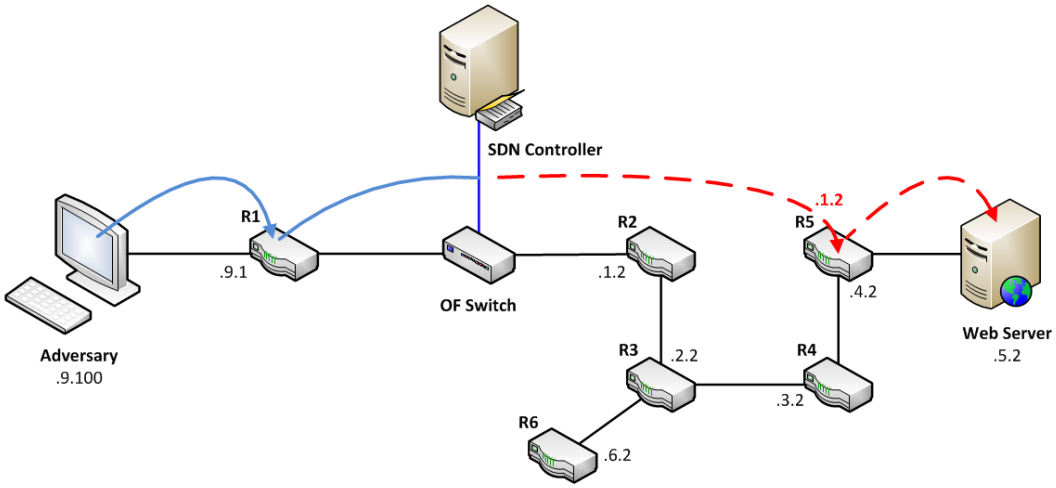

Figure 5.2: The deceptive route presented with our SDN deception. The solid blue arcs represent the true route, and the dashed red arcs represent the deceptive route provided to the adversary, in response to a traceroute probe.

5.1 General Observations

In all the tests conducted, with each ping and traceroute utility, we experienced what would be considered a significant RTT for our test environment. Specifically, we will see that the average RTT for ping tests from the adversary computer, through the normal network topology, as represented in Figure 5.1, to the actual web server and back, is about 100 milliseconds. This delay in our test environment is significant, considering that the network we are running the test are comprised of virtual routers and virtual computers, all on the same host computer, with the exception of the directly connected physical OF-enabled switch. Figure 5.3 is the actual graphic of our test environment as constructed and simulated in GNS3. All the links between nodes in the virtual environment of GNS3 are essentially software links to move traffic between virtual nodes. The inclusion of hubs in our test environment was required in GNS3 to cross-link between virtual interfaces and physical hardware interfaces, and would not be required in an actual network architecture.

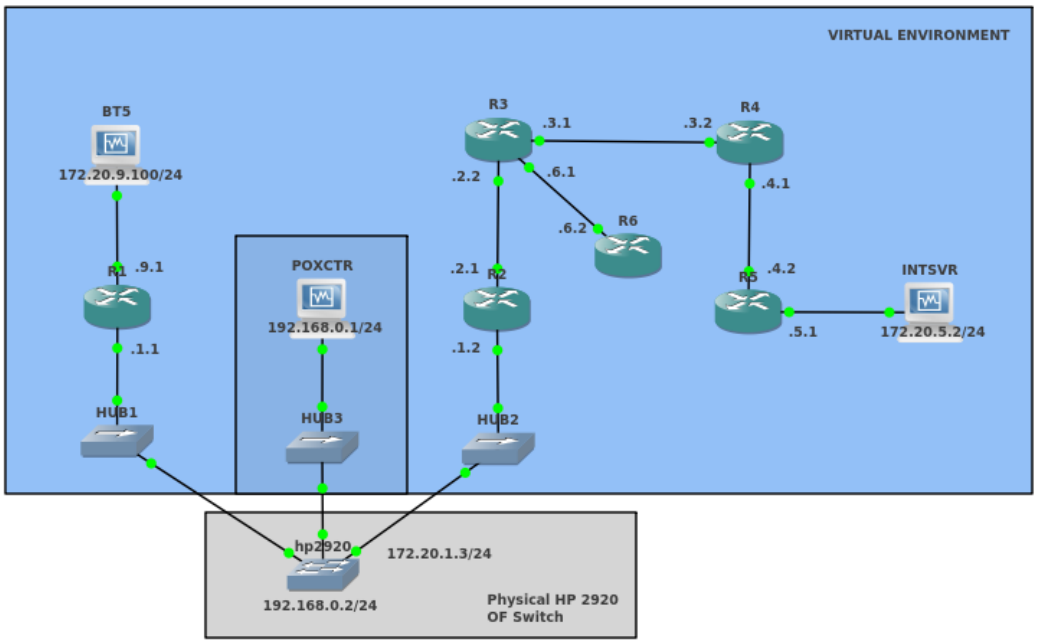

Figure 5.3: Graphic of constructed test environment in GNS3.

Recall from §2.2.3, that RTT is the duration of time it takes for a response to be received back from a probe packet. The majority of a packet's end-to-end delay, as it traverses

a typical network or the Internet, normally comes from propagation delay of the packet traveling to the destination, combined with the propagation delay of the response packet traveling back to the source. As was briefly mentioned in §2.2.3, it is this latency in packet delivery from propagation delay that is extremely useful in many latency-based geolocation methods. There are, however, other less significant factors that affect the overall end-to-end delay of packets: processing delay, transmission delay, and queuing delay. We will further expound on these four factors of end-to-end packet delay.

The speed of light in a vacuum travels at nearly 300,000 kilometers per second (186,200 miles per second), by which a signal traveling in a medium, such as copper wire or fiber optic cable, travels at about 2/3 the speed of light, or approximately 222,000 kilometers per second (138,000 miles per second) [83]. To determine the propagation delay of a signal, we simply take the distance traveled divided by the propagation speed of the signal. If we were to consider the average RTT of 100 ms for all normal probes in our end-to-end tests as previously mentioned, and assuming the other factors of processing delay, transmission delay, and queuing delay, are not a factor, then the propagation delay in one direction would be about 50 ms. This propagation delay in a real-world network would be representative of a distance of about 6900 miles in one direction; the speed of a the signal through a medium (138,000 miles per second) multiplied by the propagation delay in one direction (50 ms).

For a comparative analysis of our test measurements, consider the rough distance from the east coast to west coast of the U.S. is approximately 3,000 miles. The propagation delay, in one direction, is calculated at 3000 miles divided by 138,000 miles per second, for a total of .022 seconds (22 milliseconds). If we sent a ping from a node on one coast to a node on the other coast to solicit a reply, the RTT would be at least 44 milliseconds, or twice the propagation delay in one direction, having traveled a total of 6000 miles. Therefore, one or more of the other factors of end-to-end packet delay is contributing to the delays in our ping and traceroute measurements.

The processing delay is the time it takes for a computer or router to process the packet headers. Given that most Internet routers just inspect a packet header to determine the best path to forward the packet on, very little processing time should be involved. Though some network services, such as firewalls and proxy servers, that provide more in-depth packet inspection based on configured security policy, could provide greater processing delay.

Transmission delay is the amount of time it takes for a node to push all of a packet's bits on the medium. The two factors that impact transmission delay is the size of the packet and the data-rate of the link. Transmission delay was a significant factor in the days of dial-up networking, but is insignificant in today's high-speed networks. The last of the end-to-end factors is queuing delay, which is the time a packet waits in some buffer of a switch, router, or server, to be processed. Queuing delay can quickly become more of a factor in congested networks and especially within a congested data center.

In an effort to determine where potential delay in packets is occurring in our test environment from Figure 5.3, we conducted an experiment where we pinged each individual hop address from the adversary computer and from the web server, recording the average RTT over a thousand cycles. In this experiment, our OF-enabled switch simply functioned as a normal Ethernet switch, without any deception scheme employed. The only other traffic that transverses our test network, other than the ICMP probes, are the single packet RIP updates each router broadcast to its neighbor every 30 seconds. We included additional options with our ping probes to further limit the amount of traffic queued on the interfaces. The "-n" option was set to only output the IP address of each return so that no attempts are made to resolve to symbolic names via DNS. To also ensure that no more one unanswered probe was present on the network at a time, which could result in queuing on a node, we set the "-A" option for Adaptive ping. This also allowed immediately sending the next probe once a response was received for the previous node.

Our first test recorded the average RTT of a thousand pings from the adversary computer at 172.20.9.100 to the first hop router (R1) at 172.20.9.1. Then recording the average RTT from the adversary computer to the next hop router (R2) at 172.20.1.2. We continued this process along each hop until last recording the average RTT from the adversary computer to the web server at 172.20.5.2. We then repeated this process starting from the web server, recording the average RTT to the first hop router (R5) at 172.20.5.1. Then recording the average RTT from the web server to the next hop router (R4) at 172.20.4.1. Again, we continued this process along each hop until last recording the average RTT from the web server to the adversary computer at 172.20.9.100. Once all measurements were conducted, we plotted the average RTT recorded from each endpoint node (adversary computer and web server) to each hop along the path to the other endpoint node as represented in Figure

5.4. Other than a slight dip of a few milliseconds at hop three, router (R3), when tracing from the adversary computer, the average RTTs in both directions are essentially the same along each hop. Therefore, no one node, or the actual physical network switch, stands out as providing more delay than any other node. Given the constraints mentioned to ensure only one probe was on the network at a time, and the resulting RTTs being similar to tests later explained in §5.2, it is unlikely any delays were the result of queuing either. Also, given that all packet transmissions are mostly done through software, with the exception of crossing to and from the one physical switch, by which we also did not see any significant change in hop-to-hop delay time, that the transmission delay was also likely not a factor in our end-to-end packet delays.

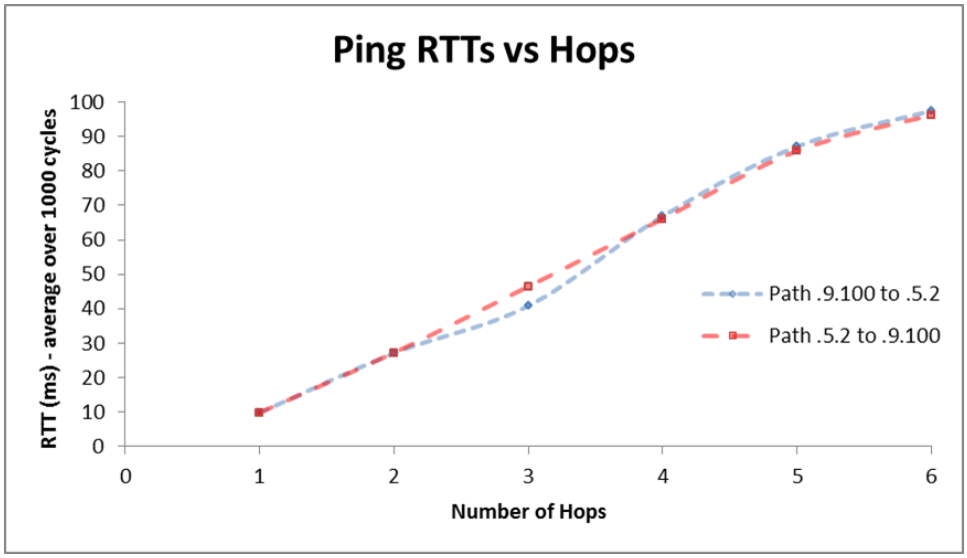

Figure 5.4: Result of delay traces in virtual test environment. The switch is functioning as a normal Ethernet switch, without any deception scheme employed.

It is therefore possible that the end-to-end delays are induced by the significant overhead of virtual processing of packets through our test environment in GNS3. The average Central Processing Unit (CPU) utilization of the virtual router emulator in our test environment averaged 30 percent, regardless of the network traffic loading placed on the network. A probe transiting through our normal network topology would have to pass through five virtual routers, each adding additional delay in processing packets. For our deceptive topology, only one virtual router is transited for probe packets, and forwarded to the controller for

processing. We will further discuss these observations in Chapter 6.

5.2 Results from PING Utilities

One of the most basic of network diagnostic tools discussed in §2.2.3 is the PING utility, which directs an ICMP Type 8, Echo message to a distant host to solicit an ICMP Type 0, Echo Reply. Most Linux based ping utilities will continue to send probes until the program is interrupted, however, for our testing purposes we limit out test to four with the "-c 4" option set in each utility tested.

DEFAULT PING. We first test the default PING utility installed on the Linux Operating System. Here we see the normal response to our ping probe in Figure 5.5, compared to the deceptive response in Figure 5.6. The normal response is similar to the general characteristics of a ping as was discussed in §2.2.3. In both, we can see that each Echo probe sent received an associated Echo reply. With each reply, the utility calculated the RTT and displays the TTL of the received Echo reply packet. Based on the discussion in §4.6 regarding default operating system TTLs, combined with other endpoint analysis tools that can be employed to fingerprint an operating system, we could assume that the target computer is running Linux or Unix, and is approximately 5 hops away. The results in Figure 5.5 are consistent with our normal network topology as represented in Figure 5.1. In the deceptive response, Figure 5.6, we see the TTL is obviously different, and applying the same assumptions, it would appear as though the target computer is only two hops away, which would be consistent with our deceptive network topology as represented in Figure 5.2.

```
# ping -c 4 172.20.5.2
PING 172.20.5.2 (172.20.5.2) 56(84) bytes of data.
64 bytes from 172.20.5.2: icmp_seq=1 ttl=59 time=97.1 ms
64 bytes from 172.20.5.2: icmp_seq=2 ttl=59 time=95.7 ms
64 bytes from 172.20.5.2: icmp_seq=3 ttl=59 time=95.3 ms
64 bytes from 172.20.5.2: icmp_seq=4 ttl=59 time=92.7 ms

--- 172.20.5.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 92.754/97.248/97.178/1.625 ms
```

Figure 5.5: Normal ICMP PING results with default Linux PING utility.

```
# ping -c 4 172.20.5.2
PING 172.20.5.2 (172.20.5.2) 56(84) bytes of data.
64 bytes from 172.20.5.2: icmp_seq=1 ttl=62 time=45.7 ms
64 bytes from 172.20.5.2: icmp_seq=2 ttl=62 time=62.9 ms
64 bytes from 172.20.5.2: icmp_seq=3 ttl=62 time=22.4 ms
64 bytes from 172.20.5.2: icmp_seq=4 ttl=62 time=10.5 ms

--- 172.20.5.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 10.528/35.402/62.953/20.330 ms
```

Figure 5.6: Deceptive ICMP PING results with default Linux PING utility.

FPING. We now test the Fping utility against our target computer. Recall from §2.2.3 that the Fping utility is similar to the standard Linux installed ping utility, but was developed to enable pinging more than one computer at a time and enabled improved scripting functionality. Fping is limited to only using ICMP for constructing network probes. Here we see the default response to our ping probe in Figure 5.7, compared to the deceptive response in Figure 5.8. As in the default ping utility, we can see in both the normal and deceptive topology, that each Echo probe sent received an associated Echo reply. However, the output is somewhat different than the standard ping utility. Specifically, the output does not provide the TTL or ICMP sequence number of the response packets. In addition to displaying the RTT for each response, it calculates and displays the running RTT average. On the surface, the Fping utility does not provide any better detail than the standard ping utility.

```
# fping -c 4 172.20.5.2
172.20.5.2 : [0], 84 bytes, 92.8 ms (92.8 avg, 0% loss)
172.20.5.2 : [1], 84 bytes, 92.4 ms (92.6 avg, 0% loss)
172.20.5.2 : [2], 84 bytes, 95.6 ms (93.6 avg, 0% loss)
172.20.5.2 : [3], 84 bytes, 98.4 ms (94.8 avg, 0% loss)

172.20.5.2 : xmt/rcv/%loss = 4/4/0%, min/avg/max = 92.4/94.8/98.4
```

Figure 5.7: Normal ICMP PING results with the Fping utility.

```
# fping -c 4 172.20.5.2
172.20.5.2 : [0], 84 bytes, 39.5 ms (39.5 avg, 0% loss)
172.20.5.2 : [1], 84 bytes, 25.8 ms (32.7 avg, 0% loss)
172.20.5.2 : [2], 84 bytes, 37.3 ms (34.2 avg, 0% loss)
172.20.5.2 : [3], 84 bytes, 55.8 ms (39.6 avg, 0% loss)

172.20.5.2 : xmt/rcv/%loss = 4/4/0%, min/avg/max = 25.8/39.6/55.8
```

Figure 5.8: Deceptive ICMP PING results with the Fping utility.

HPING. Next, we use the Hping utility against our target computer. We briefly mentioned the Hping utility in §2.2.3 when discussing various ping utilities. A key distinction from the default ping and Fping utility, is that Hping can perform probes with TCP, UDP, Raw-IP, in addition to ICMP. As such, we add an additional option to specify use of ICMP with the "-1" option. Here we see the default response to our ping probe in Figure 5.9, compared to the deceptive response in Figure 5.10. The output of ICMP Hping probes is nearly identical to the output of the default ping utility in Figure 5.5. With each reply, the utility calculated the RTT and displays the TTL of the received Echo reply packet.

Hping also presents the "id" field from the IP header of the Echo reply packet, known as the Internet Protocol Identification (IPID). Recall in §3.4, the IP header as represented in Figure 3.3, which represents this field as the identification field. This field is used for datagram fragmentation and reassembly, but is often employed by tools for fingerprinting. Every computer and router implements an internal counter, which is incremented for each packet sent generated and sent. In the normal topology, it would appear that the end host increments the IPID field by a value of one for each packet. In the deceptive topology, the end host appears to increment the identification by a value of two for each packet. Though this incremental value by two as compared to the normal topology response does appear a bit odd, the adversary would only receive the deceptive response with the deception scheme employed, and therefore may not appear significant. The results in Figure 5.9 are consistent with our normal network topology as represented in Figure 5.1. And in the deceptive response, Figure 5.10, we see the TTL is consistent with the results in Figure 5.6, and with our deceptive network topology as represented in Figure 5.2.

```
# hping3 -c 4 -1 172.20.5.2
HPING 172.20.5.2 (eth1 172.20.5.2) icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.20.5.2 ttl=59 id=17038 icmp_seq=0 rtt=108.7 ms
len=46 ip=172.20.5.2 ttl=59 id=17039 icmp_seq=1 rtt=98.5 ms
len=46 ip=172.20.5.2 ttl=59 id=17040 icmp_seq=2 rtt=103.9 ms
len=46 ip=172.20.5.2 ttl=59 id=17041 icmp_seq=3 rtt=96.1 ms

--- 172.20.5.2 hping statistic ---
4 packets transmitted, 4 received, 0% packet loss
round-trip min/avg/max/ = 96.1/101.8/108.7 ms
```

Figure 5.9: Normal ICMP PING results with the Hping utility.

```
# hping3 -c 4 -1 172.20.5.2
HPING 172.20.5.2 (eth1 172.20.5.2) icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.20.5.2 ttl=62 id=48619 icmp_seq=0 rtt=26.0 ms
len=46 ip=172.20.5.2 ttl=62 id=48621 icmp_seq=1 rtt=47.3 ms
len=46 ip=172.20.5.2 ttl=62 id=48623 icmp_seq=2 rtt=27.1 ms
len=46 ip=172.20.5.2 ttl=62 id=48625 icmp_seq=3 rtt=47.6 ms

--- 172.20.5.2 hping statistic ---
4 packets transmitted, 4 received, 0% packet loss
round-trip min/avg/max/ = 26.0/37.0/47.6 ms
```

Figure 5.10: Deceptive ICMP PING results with the Hping utility.

NPING. Last, we employ the Nping utility against our target computer. Like the Hping utility, Nping can perform probes with TCP, UDP, and Raw-IP, in addition to ICMP. Therefore, we must also specify in the command syntax the protocol option to use, by which we pass Nping the "–icmp" option for our testing purposes. Here we see the default response to our ping probe in Figure 5.11, compared to the deceptive response in Figure 5.12. Recall from 2.2.3, Nping has a unique feature called "Echo Mode" to provide advanced troubleshooting and discovery so users can see how packets change in transit without the use of packet capture utilities for in-depth packet comparison. Part of this "mode" is represented by the addition of not only information about the received packet, but also some information regarding the probe packet that solicited the response packet.

Clearly the output format is very different than that of the other utilities used. Nping, much like Hping, has a long list of selectable options to customize and tweak network probe packets as it is more in-tune for network penetration testing than just a simple network diagnostic utility. Like Hping, Nping also presents the "id" field from the IP header of the Echo reply packet. In the normal topology, it would appear that the end host increments the identification field by a value of one for each packet. In the deceptive topology, the end host appears to increment the identification by a value of two for each packet. We saw similar results with the Hping utility as well. In addition, we can see Nping uses the same "ipid" for each probe message sent to the target, and therefore, would not use this field for uniquely identifying individual probe messages.

Nping also has additional options to provide even further details than what is represented below. In fact, the default output line was so long, that we truncated a portion of its output to fit the width of the page. The "<t/c>" in each SENT and RCVD line represents the reg-

istered ICMP Type and Code from the packets in the probes. In both normal and deceptive topology probes, all the SENT probe's "<t/c>" values were (type=8/code=0), for a typical Echo request; and all the RCVD responses "<t/c>" values were (type=0/code=0), for a typical Echo reply. On further inspection, the overall information obtained from an Nping probe is the same as the other utilities, only that Nping displays more of the underlying packet information to the user. We see in Figure 5.11 and Figure 5.12, that the TTLs and RTTs are similar to the default ping and Hping utilities, and consistent with the network topology in Figure 5.1 and Figure 5.2, respectfully.

```
# nping -c 4 --icmp 172.20.5.2

Starting Nping 0.5.51 ( http://nmap.org/nping ) at 2014-02-28 08:51 PST
SENT (0.0015s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=65389 iplen=28
RCVD (0.1048s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=59 id=17042 iplen=28
SENT (1.0022s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=65389 iplen=28
RCVD (1.0946s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=59 id=17043 iplen=28
SENT (2.0044s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=65389 iplen=28
RCVD (2.1044s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=59 id=17044 iplen=28
SENT (3.0075s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=65389 iplen=28
RCVD (3.1074s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=59 id=17045 iplen=28

Max rtt: 103.187ms | Min rtt: 92.164ms | Avg rtt: 98.340ms
Raw packets sent: 4 (112B) | Rcvd: 4 (184B) | Lost: 0 (0.00 %)
Tx time: 3.00627s | Tx bytes/s: 37.26 | Tx pkts/s: 1.33
Rx time: 4.00836s | Rx bytes/s: 45.90 | Rx pkts/s: 1.00
Nping done: 1 IP address pinged in 4.01 seconds
```

Figure 5.11: Normal ICMP PING results with the Nping utility.

```
# nping -c 4 --icmp 172.20.5.2

Starting Nping 0.5.51 ( http://nmap.org/nping ) at 2014-02-28 08:54 PST
SENT (0.0014s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=41803 iplen=28
RCVD (0.0377s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=62 id=48824 iplen=28
SENT (1.0019s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=41803 iplen=28
RCVD (1.0632s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=62 id=48826 iplen=28
SENT (2.0033s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=41803 iplen=28
RCVD (2.0349s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=62 id=48828 iplen=28
SENT (3.0051s) ICMP 172.20.9.100 > 172.20.5.2 Echo request <t/c> ttl=64 id=41803 iplen=28
RCVD (3.0578s) ICMP 172.20.5.2 > 172.20.9.100 Echo reply <t/c> ttl=62 id=48830 iplen=28

Max rtt: 61.13ms | Min rtt: 31.398ms | Avg rtt: 45.238ms
Raw packets sent: 4 (112B) | Rcvd: 4 (184B) | Lost: 0 (0.00 %)
Tx time: 3.00402s | Tx bytes/s: 37.28 | Tx pkts/s: 1.33
Rx time: 4.00562s | Rx bytes/s: 45.94 | Rx pkts/s: 1.00
Nping done: 1 IP address pinged in 4.01 seconds
```

Figure 5.12: Deceptive ICMP PING results with the Nping utility.

5.3 Results from Traceroute Utilities

The various ping utilities we tested, all represented our normal and deceptive topology as expected in Figure 5.1 and Figure 5.2, respectfully. However, since we are only sending probes to and getting responses from an endpoint node, we can only infer the number of hops between our test node and the endpoint node. To gain detailed knowledge of the actual intermediate nodes along the path to the target endpoint node, we must use various traceroute utilities as discussed in §2.2.4. Again, since our focus is on ICMP, we will send ICMP Type 8, Echo messages to solicit ICMP Type 0, Echo Replies. Traceroute utilities differ from ping utilities by manipulating the TTL to solicit ICMP Type 11, Code 1, Destination Unreachable / Time Exceeded in Transit, messages from those intermediate nodes, or routers, along the path to our target.

Traceroute utilities, like ping utilities, provide many different options to manipulate the way in which the probe messages are constructed, and how the results are output to the user. A typical default function of traceroute utilities is to attempt to resolve the IP address of each hop to a Fully Qualified Domain Name (FQDN). Since our test environment does not employ DNS services to perform this resolution, we include the "-n" option to prevent this function. This also does not impact the deception scheme, though inclusion of deceptive DNS entries could further enhance the believability of the deceptive scheme. Linux and Unix traceroute utilities typically use UDP by default for performing traces, unless otherwise specified with various options. Therefore, include the "-I" option with the default traceroute utility, and "-p icmp" for the paris-traceroute utility, to specify use of ICMP for all traces. As a reminder when review the traceroute tests, since our goal is presenting a deceptive topology to network probes, the issues with RTT in our test environment as discussed in §5.1.

DEFAULT TRACEROUTE. We first test the default Linux traceroute utility installed on the Adversary computer in our test environment. Here we see the normal response to our trace in Figure 5.13, compared to the deceptive response in Figure 5.14. The normal response is similar to the general characteristics of a trace as was discussed in Figure 2.2 in §2.2.4. With each set of three probes, sent at each TTL value starting with one, we are able to discover the individual router hops along the path to the destination. With each TTL value, the default traceroute program also displays the calculated RTT of each individual

probe. The results in Figure 5.13 for the normal topology is consistent with the number of hops inferred from our previous ping tests, and is representative of the normal topology illustrated in Figure 5.1. The target of our traceroute is exactly five hops away, with the sixth hop being the target node itself. The results in Figure 5.14 for the deceptive topology also confirms the number of hops inferred from previous ping tests, and is representative of the deceptive topology illustrated in Figure 5.2. The target of our traceroute, through the deception employed, is presented to the adversary as being precisely two hops away, with the third hop being the target node itself.

```
# traceroute -n -I 172.20.5.2
traceroute to 172.20.5.2 (172.20.5.2), 30 hops max, 60 byte packets
 1  172.20.9.1  4.208 ms  14.621 ms  25.879 ms
 2  172.20.1.2  35.194 ms  45.171 ms  55.203 ms
 3  172.20.2.2  75.661 ms  85.658 ms  95.785 ms
 4  172.20.3.2  125.983 ms  136.170 ms  146.508 ms
 5  172.20.4.2  179.490 ms  187.817 ms  198.527 ms
 6  172.20.5.2  218.498 ms  226.512 ms  226.185 ms
```

Figure 5.13: Normal ICMP Traceroute results with default traceroute utility.

```
# traceroute -n -I 172.20.5.2
traceroute to 172.20.5.2 (172.20.5.2), 30 hops max, 60 byte packets
 1  172.20.9.1  3.322 ms  13.114 ms  23.663 ms
 2  172.20.1.2  33.626 ms  43.575 ms  53.277 ms
 3  172.20.5.2  63.661 ms  73.182 ms  84.500 ms
```

Figure 5.14: Deceptive ICMP Traceroute results with default traceroute utility.

PARIS-TRACEROUTE. We next test the paris-traceroute utility that was briefly mentioned in §2.2.4, and further discussed in §2.3.1. Here we see the normal response to our trace in Figure 5.15, compared to the deceptive response in Figure 5.16. Again, with each set of three probes, sent at each TTL value starting with one, we are able to discover the individual nodes along the path to the destination. And, with each TTL value, paris-traceroute displays the calculated RTT of each individual probe. The results in Figure 5.15 for the normal topology is similar to the results from the default traceroute utility in Figure 5.13, and hence is representative of the normal topology illustrated in Figure 5.1. As well, the results in Figure 5.16 for the deceptive topology is similar to the results from the default traceroute utility in Figure 5.14, and hence is representative of the deceptive topology illustrated in Figure 5.2.

```
# paris-traceroute -n -p icmp 172.20.5.2
traceroute [(172.20.9.100:33456) -> (172.20.5.2:33457)], protocol icmp, duration 1 s
 1  172.20.9.1  2.356 ms 3.159 ms 2.174 ms
 2  172.20.1.2  22.224 ms 22.861 ms 22.595 ms
 3  172.20.2.2  44.080 ms 44.065 ms 45.204 ms
 4  172.20.3.2  65.081 ms 66.364 ms 65.341 ms
 5  172.20.4.2  90.946 ms 90.890 ms 90.899 ms
 6  172.20.5.2  101.326 ms 101.330 ms 102.55 ms
```

Figure 5.15: Normal ICMP Traceroute results with paris-traceroute utility.

```
# paris-traceroute -n -p icmp 172.20.5.2
traceroute [(172.20.9.100:33456) -> (172.20.5.2:33457)], protocol icmp, duration 1 s
 1  172.20.9.1  5.787 ms 5.686 ms 5.479 ms
 2  172.20.1.2  35.811 ms 29.301 ms 49.289 ms
 3  172.20.5.2  28.939 ms 49.726 ms 29.936 ms
```

Figure 5.16: Deceptive ICMP Traceroute results with paris-traceroute utility.

5.4 General Assessment of Deception Scheme

On the surface, our deception scheme employed through SDN and the OF protocol appear to present a believable deceptive topology. The various ping and traceroute utilities run against the test network with ICMP all appear to accurately map the normal network topology. Additionally, these utilities all generally appear to accurately represent the intended deceptive topology employed though our deception scheme. Therefore, it would appear that employing our scheme in a normal network environment, avoid of the RTT issues from the virtual environment, would deceive an adversary as to the true topology of the network when employing ICMP-based utilities.

CHAPTER 6:
CONCLUSION AND FUTURE WORK

In this thesis, we demonstrated that SDN and the OF protocol provide several benefits to employing topology deception through a proof-of-concept SDN network controller. For example, the centralized controller permits agility. Any changes made to the deception scheme are easily implemented in the SDN controller and deployed through a simple restart, thereby pushing the new policy to the network. No software or hardware upgrades are required to dynamically change the deception functionality. However, our solution is still a work-in-progress and many issues remain to be resolved to move towards a full-scale, ready-to-deploy, enterprise solution. This chapter details some of the weakness in the overall deception scheme, provides some potential alternative approaches to employing the deception, and discusses potential avenues for future work.

6.1 Weaknesses in Deception Scheme

There are certainly many weaknesses in our present deception implementation, some of which are fundamental and some are implementation-specific. We further acknowledge that some weaknesses may remain unidentified in this thesis. For those identified, some will be easier to correct than others. To move towards construction of a more robust, enterprise-grade deceptive topology solution, these weaknesses must be addressed. We will discuss two of these major weaknesses, one discovered from previous work [16] and one more inherent in the current standards of the OF protocol.

6.1.1 Weakness of a Central Node

In §2.3.2, we discussed some general differences between low-interaction and high-interaction honeypots. Low-interaction honeypots generally only provide some basic level of faked services for intentional deception. Further investigation by an experienced adversary, employing the right tools against low-interaction honeypots, and with the correct analysis, could uncover that some type of deception scheme was employed.

The initial work [16] on deceptive dynamic network topology that we are building on could still be considered a type of low-level deception scheme. The work discussed some criteria

that the topological deception must satisfy to be successful; the first is that the deception presented must be "believable." From the results of our test in Chapter 5, using four different ping utilities and two different traceroute utilities, the deception is believable and easily deceives large-scale automated probing. Security researchers, research groups, on the other-hand would probably be able to figure out the deception. And an adversary who is focused particularly on gaining access to and exploiting the defended network, would quickly uncover the current deception scheme through more detailed, focused scans and analysis.

In all the scans we conducted with the various tools, we only employed the default options to get our results. But as previously mentioned, many of these tools have additional options that can be set to further manipulate the probe packets, or display more details of the results from the probes. The paris-traceroute tool has a "-i" option to display the IPID for each individual probe when conducting a scan. Some of the ping utilities tested in §5.2, also presented the IPID with each probe, but the results from those ping tests are consistent with normal responses from a single node, even with the deceptive topology.

In Figure 6.1, we show the same scan with the paris-traceroute utility as was illustrated in Figure 5.15, but this time set the option to display the IPID of each probe displayed. To an inexperienced person, the below results may not seem unusual, however, to someone experienced in network traffic analysis, the results shown below would seem unusual. As mentioned in §5.2, each router maintains an internal counter that is incremented for each packet it originates, but does not count those it forwards [84]. This includes replies to pings and traceroutes, sending routing protocol route updates to neighbor routers, providing flow data for network management, and for interactive terminal access by administrators.

```
# paris-traceroute -n -i -p icmp 172.20.5.2
traceroute [(172.20.9.100:33456) -> (172.20.5.2:33457)], protocol icmp, duration 1 s
 1  172.20.9.1  2.356 ms {334} 3.159 ms {335} 2.174 ms {336}
 2  172.20.1.2  22.224 ms {275} 22.861 ms {276} 22.595 ms {277}
 3  172.20.2.2  44.080 ms {401} 44.065 ms {402} 45.204 ms {403}
 4  172.20.3.2  65.081 ms {277} 66.364 ms {278} 65.341 ms {279}
 5  172.20.4.2  90.946 ms {276} 90.890 ms {277} 90.899 ms {278}
 6  172.20.5.2  101.326 ms {17077} 101.330 ms {17078} 102.55 ms {17079}
```

Figure 6.1: Normal ICMP Traceroute results with paris-traceroute utility, adding the option to display IPID.

It would seem unlikely for several routers in a network to all have near identical IPID numbers, though unique conditions could establish such results. In our test network, all the virtual routers are from the same vendor IOS, and hence all started up at the same time when the test network was brought online. Adding the fact that we are not introducing any additional traffic into the network beyond our normal end-to-end ping and traceroute tests, there is little chance for the router IPIDs to diverge. The probability of an actual production network exhibiting similar indications is likely very low, unless a group of routers in a network was recently restarted, and that vendor's product initializes the counter for the IPID from a deterministic low number, or zero, vice a random number. However, even is such a scenario, those routers' IPID would probably diverge quickly.

The results of the paris-traceroute probe with the IPID option set, as illustrated in Figure 6.2, with our deception scheme employed, is even more interesting. The results of this scan would quickly raise a red flag with an experienced network traffic analyst or experienced hacker. The probability of a host computer, when conducting a trace to that host, exhibiting an IPID that immediately follows the same incrementing pattern sequence as the router one hop previous, is highly unlikely. Such results would likely attract further scans and probes in attempts to fingerprint each node, and could tip an adversary that some form of deception was employed, as it would appear that the node at hop two and three were the same.

```
# paris-traceroute -n -i -p icmp 172.20.5.2
traceroute [(172.20.9.100:33456) -> (172.20.5.2:33457)], protocol icmp, duration 1 s
 1  172.20.9.1  5.787 ms {416} 5.686 ms {417} 5.479 ms {418}
 2  172.20.1.2  35.811 ms {51152} 29.301 ms {51154} 49.289 ms {51156}
 3  172.20.5.2  28.939 ms {51158} 49.726 ms {51160} 29.936 ms {51162}
```

Figure 6.2: Deceptive ICMP Traceroute results with paris-traceroute utility, adding the option to display IPID.

To better evaluate these two IP addresses, an adversary can employ alias resolution techniques in attempts to determine if they are the same node. According to Keys [85], there are generally two ways of classifying alias resolution techniques: fingerprinting techniques and analytical techniques. The most effective method to further exploit the unusual observations with the IPID in Figure 6.2, is to use an analytical technique to better determine if they are from the same node. According to Keys, two addresses can be inferred to be aliases if they have similar and constant IP velocities, and the ID value in every response

from one address is similar to the interpolated ID value of the other address at the same time. This is because a multi-homed server or a router will typically use a single IPID counter shared by all ports. Though from an adversarial standpoint, determining with a high degree of confidence that these two IP address are from the same node, would still be quite confusing considering one is apparently a web server and the other represented as a router. The web server would easily be identified by DNS records, though many network routers also have associated DNS records to identify its FQDN. Implementing deceptive DNS entires would complement the overall deceptive scheme to make it more believable to an adversary.

To conduct our analytical test of these two IP address, we use a simple script that alternates pinging each IP address over a sufficient duration of time or number of iterations, to develop enough data to plot and evaluate the results. For our specific script, we simply alternate pinging both hosts over an iterative period of one thousand times. We run a packet capture utility to record all the probe messages and associated responses. We then export from the capture file ICMP response messages with relative time and IPID. From the filtered results, we can plot the IPID velocity versus time for both IP addresses as represented in Figure 6.3. Here we see that the IPID velocities for both IP address have the same vectors. We saw from

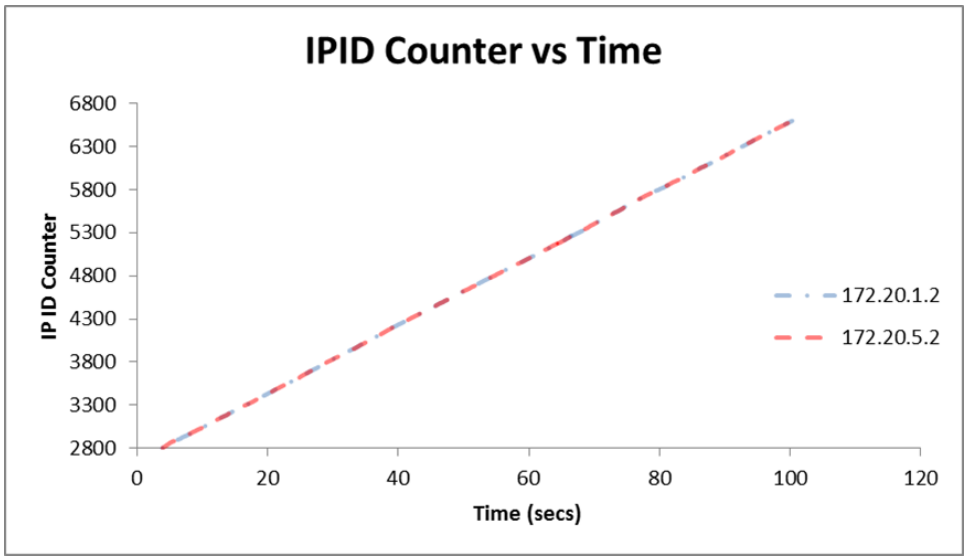

Figure 6.3: Result of IPID testing to deceptive nodes.

Figure 6.2, that the IPID in the trace at hop two and three, incremented in even numbers by a value of two. The underlying data captured for generating Figure 6.3 precisely followed this same incremental sequence. Given the results of this experiment, an adversary would easily conclude, with a high degree of confidence, that the IP address 172.20.1.2 and 172.20.5.2 were logically associated in some way with the same node, which is odd, since traceroute indicated that they were different hops, potentially revealing the deception.

In our initial SDN deception design, we did not at first consider the need of obfuscating IPID responses. It was not until testing the paris-traceroute utility with additional display options, did we discover this issue. However, this discovery could technically be performed from any ping or traceroute utility, and manually conducting a in-depth review of the captured results, and eventually noting the IPID of the responses. Therefore, to obviously present a higher level deception scheme, future work will require implementing additional methods to control use of IPIDs in deceptive nodes, to prevent discovery of a central control node.

6.1.2 Weakness of OpenFlow

We discussed SDN and the OF protocol at length in Chapter 3. In §3.4, we specifically discussed the many details of the OF switch specification to include the fields that OF matches can be constructed with to set flow policy on a switch. Our deceptive implementation initially focused on ICMP probes as we only have to conduct a match on the protocol number for ICMP traffic to flag packets for processing.

Previous work [16] focused initially on providing deceptive responses to UDP probe traffic. Any robust deceptive topology solution employed would have to effectively provide deceptive responses to both ICMP and UDP probe traffic. In addition, it must correctly identifying TCP probe traffic over legitimate TCP traffic, and provide deceptive response to the TCP probes as well. Recall from §2.2.4, that UDP ports 33434 through 33534 are IANA registered for traceroute use. Traceroute utilities that use UDP to conduct a trace, increment the port number starting from 33434 for each probe packet sent. To screen UDP packets for the presence of normally constructed UDP probes, compared to UDP packets carrying legitimate network traffic, we have 101 separate port numbers to also conduct a match for, in addition to matching for the UDP protocol. The challenge in this, as also

discussed in §3.4, is that we cannot specify a transport port range in a flow rule by current OF switch standards. Therefore, to screen for UDP probes for processing and providing deceptive responses, we must create 101 separate flow rules, one for each port number 33434 through 33534. This should not be too difficult to construct the rule sets with OF for particularly matching on the traceroute UDP ports. A simple "for" loop, starting from the first port number of 33434, and iterating through to port number 33534, can be used to set each flow rule on the switch. There are no dedicated port numbers for TCP probes, as is with UDP probes, which presents a challenge on how to differentiate TCP probes from legitimate TCP network traffic?

Another weakness with OF with respect to our ability to implement deception, is that we must send all identified probes to the controller for processing and generation of deceptive responses. As we previously discussed, a typical router will screen an incoming packet's TTL, and if that packet's TTL is equal to one and not destined for a port on that router, then the TTL is decremented, the packet dropped, and an ICMP Time Exceeded message is generated from the router and sent back to the source of the dropped packet. The OF protocol does not provide the ability generate conditional response messages directly from an OF-enabled switch. Most limiting with OF in regards to deceiving topology mapping from various traceroute utilities, is the inability to match on TTL values, as expiring TTLs is what triggers the needed message for traceroute to build its trace.

6.2 Alternate Approaches

There are potentially numerous different approaches that can be taken in implementing a dynamic deceptive network topology, beyond what we have presented here. Some alternate approaches may be more practical and effective than others. In high traffic networks, continued work at quickly processing large amounts of packets at the kernel level, vice user level with an SDN controller, may potentially be a better approach. Others may argue for implementing the deception with an entire virtual overlay network, however, such an implementation could incur the same delay issues as our test network had, as discussed in §5.1. We advocate another potential option that can continue to build on the initial work from this thesis in employing an SDN solution.

Any robust deceptive topology solution employed would have to effectively provide deceptive responses to both ICMP, UDP and TCP probe traffic. Regardless of the protocol type of incoming probe traffic, they all would solicit a limited range of ICMP responses. A better alternate approach may be to simply allow all probe traffic to enter the network, and instead filter and manipulate the outgoing ICMP responses to those probes to present our deceptive topology. The potential challenge in this approach is that we could not implement a deception scheme that presents more fake hops to a network resource than the number in the true path. An adversary could simply compare the response times from legitimate service request to the protected server, compared to that of various traces of the network path to the server. If the server responses are faster than the responses along the path to the server, then the adversary would learn that some form of deception was in use.

6.3 Future Work

We highlighted some of the weaknesses and potential alternate approaches to the work conducted in this thesis. Some of these must be addressed to provide a more robust deceptive topology scheme. Other weakness, such as the limited field match options, may simply never change in future OF switch standards, requiring working within those limits. All of which provides room for additional development, and there are many other potential areas for future development to this work as well. Any deceptive topology solution must take into consideration the wide range of network mapping efforts, some of which was discussed in Chapter 2.

6.3.1 Building upon the POX SDN Controller

We briefly mentioned some of the challenges and limitations of our methodology in Chapter 3. The code written to implement our POX SDN OF controller was done with a focus towards functionality in developing a proof-of-concept, and not with efficiently in mind. Many of the variables and the IP addresses of our deceptive nodes were all hard coded instead of taking a more modular approach to programming. We also only implemented one level of depth in our deceptive topology, which provided some level of complexity in following our code. To follow the same method used implementing additional depth in the deceptive topology with more routers, would be highly inefficient. In §3.2, we discussed the concept of logical layers in the SDN architecture, in which applications would sit on top

of the SDN controller itself. Future work could further modularize our custom controller to handle the basics of packets in with the controller, and through an API, interface the deception topology application module to scan the actual network topology, and build an optimized deceptive topology from that scan.

As discussed in §5.1, we observed significant RTTs from running our controller in a completely virtualized test network, that would not be representative of an actual network. As such, we did not code in any additional delays in the sending of deceptive responses to probe messages. This presents a weakness in our specific implementation that must be addressed along with the issue of a common IPID as discussed in §6.1.1. To further make the deception scheme more believable to adversary probes, additional response packet delay times must be used that are consistent with the intended depth representation of the deceptive topology. To then validate the model of the deception scheme, future researchers may consider building a test network from completely separate physical equipment to eliminate any potential induced issues from a virtual test environment.

6.3.2 Other SDN Controllers

Our choice of SDN controller, as discussed in §4.4, was primary due to our familiarity with the Python programming language and the ease of implementing the POX controller. The POX controller library, though limited to OF switch specification 1.0.0, with some extensions to enable certain features of higher level specifications, is fairly easy to work with. Its development is still community-driven with a focus towards research and education purposes. However, with limited community work in seeking to further the development of the POX library, its potential for use in constructing high grade enterprise SDN controller applications is also limited. After having done some development with the POX SDN controller, and while gathering additional background information on the scope of SDN controllers, we came across the Ryu [86] python-based controller, also listed in Table 3.1. The Ryu SDN framework claims to support all OF switch specifications, including the latest version 1.4.0 [87]. This latest version has expanded the OF matchable fields from the initial 12 in the first specification, to a total of 41 OF matchable fields. Researchers with a preference for the Python programming language and desiring to further explore the latest features of the OF protocol to implement a deceptive topology, may wish to use the Ryu SDN controller. Though the latest version of the OF specification still does not include the

TTL field of an IP packet as a matchable field.

As shown in §3.3, there are many other SDN controllers in various stages of development, with various levels of support. Different SDN controllers are written in different programming languages, and some may offer access to a richer set of library classes for developing a custom SDN controller for topology deception than others. As well, some SDN controllers have major vendor backing that could provide a more matured baseline to build a modularized deception topology application on top of, and a larger community of support for help in development. Additionally, some of these controllers may support newer OF switch specifications, from which some of the major changes, as highlighted in Appendix A, could enable improved functionality or implementation methods for presenting a deceptive topology through SDN and the OF protocol. The latest switch firmware release, version WB.15.14.0002, that we loaded into on our switch in our test network, as described in §4.5, added support for OF Switch Specification 1.3.0 [88]. It would be interesting to see if any of these changes in the OF switch specification can be exploited to further improve on the development of a deceptive topology SDN controller.

6.3.3 UDP and TCP Probes

As we mentioned several times throughout this thesis, any robust solution must be able to provide deceptive responses to ICMP, UDP and TCP probes. Though we highlighted a weakness of the OF protocol in §6.1.2 with regard to limits on matching UDP probes, it is not impossible to implement deception to UDP probes. Though much more efficient methods of coding the custom SDN controller as mentioned in §6.3.1 should be considered, regardless of the choice of baseline SDN controller to build from. The largest challenge may be in properly detecting TCP probes, as they can be constructed the same as normal TCP traffic.

6.3.4 Multi-Ingress Networks

Given that an SDN controller is capable of managing more than one OF-enabled switch, implementation in a multi-ingress network is not entirely impossible. Since much remains to be resolved in just providing a believable deceptive topology at a single ingress point, and doing so with SDN and the OF protocol, we chose to narrow our scope. Therefore, we focused our efforts in first implementing a solution for a single ingress point, which can

later be improved upon and scaled to protect a multi-ingress network.

However, for a deceptive topology to be completely believable, the results of probes through each ingress point of the network to be protected must complement each other. One method could be to separately implement a stand-alone solution at each ingress point. But not fully implementing an SDN solution with a centralized management node to simultaneously manage the deceptive policy at each ingress point, fails to fully take advantage of the basic premises of SDN and the OF protocol. Additional challenges must be overcome in deploying such a distributed solution if the various ingress points are geographically separated, in which the consideration of controller placement must be evaluated for the deceptive solution. If we have multiple ingress points to a large network, say in Washington DC, San Antonio, TX, and San Diego, CA, and if we centralize a single controller in St. Louis, MO, all probe packets must be sent to the centralized controller for processing, increasing the overall time to provide a deceptive reply to a probe. Therefore, for a centralized management solution, we would need to create a tiered controller structure, where a master controller directs policy to regional controllers at each ingress point. Some previous research on The Controller Placement Problem [89] has already generally explored the issue of geographically separated nodes, and could be a good starting point for researchers who wish to expand a deceptive solution to manage a multi-ingress network. As work matures on efficiently implementing deceptive dynamic network topology through SDN and with OF protocol, researchers should strive to ensure a solution capable of scaling to protect large distributed networks with multiple ingress points.

APPENDIX A:
[OpenFlow Switch Specifications]

Specification Version (date)	**Major changes**
1.0.0 (Dec. 31, 2009)	- multiple queues per output port - flows use and referenced by opaque identifier (cookie) - include switch description field (OFPST_DESC) - match IP fields in ARP packets - match IP ToS / DSCP bits
1.0.1 Errata (Jun. 7, 2012)	- clarify: table-miss actions - clarify: switch port enumeration must start with port number 1 - clarify: no padding of error messages - clarify: properly ignoring fields in packet match - clarify: use of flow removed messages - clarify: virtual ports not allowed for input port - clarify: fail-secure and fail-standalone modes of operation
1.0.2 Errata RC1 (Oct. 4, 2013)	- TCP port changed from 6633 to 6653 for switch-to-controller transport OF communication (note: TCP 6633 was never registered with the IANA for OF use and was allocated by IANA for other services)
1.1.0 (Feb. 28, 2011)	- support for pipelining with multiple tables - instructions in flow entries to control pipeline processing - allows multiple options for table-miss - reference group of ports for multicasting - improved VLAN support - add Multiprotocol Label Switching (MPLS) support - change port number to 32 bits to support virtual ports - maskable datalink and network address match fields - add TTL decrement, set and copy actions for IPv4 and MPLS - add ECN action

1.2 (Dec. 5 2011)	- IPv6 support added - change many static fields to a Type-Length-value (TLV) format, called OpenFlow Extensible Match (OXM) - numerous other extensions for more granular control
1.3.0 (Jun. 25, 2012)	- improved description of table capabilities - further changed table-miss behavior - added IPv6 extension header handling support - per controller connection event filtering - auxiliary switch-to-controller connections - numerous other extensions for more granular control
1.3.1 (Sept. 6, 2012)	- improved OF version negotiation for switch-to-controller connections - numerous modifications and clarifications to extensions
1.3.2 (Apr. 25, 2013)	- allows connection initiation from a controller - several modifications and clarifications to extensions
1.3.3 (Dec. 18, 2013)	- updated with IANA registered TCP port 6653 - extensive list of modifications and clarifications to extensions
1.4.0 (Oct. 15, 2013)	- further conversion of static fields to a TLV format as part of OXM - more descriptive reasons for packet-in - properties to differentiate optical port properties for fiber optics - flow monitoring by a controller - improved role status events for multi-controller configurations - several additional error handling types and codes

APPENDIX B:
[Switch Configuration File]

```
#HP-2920-24G# show run

; J9726A Configuration Editor; Created on release #WB.15.14.0002
; Ver #04:08.f3.35.0d:39

hostname "HP-2920-24G"
module 1 type j9726a
interface 1
   speed-duplex 100-full
   exit
interface 2
   speed-duplex 100-full
   exit
snmp-server community "public" unrestricted
openflow
   enable
   controller-id 1 ip 192.168.0.1 port 6653 controller-interface oobm
   instance "oflab"
      listen-port 6653 oobm
      member vlan 100
      controller-id 1
      connection-interruption-mode fail-standalone
      enable
      exit
   enable
   exit
oobm
   ip address 192.168.0.2 255.255.255.0
   ip default-gateway 192.168.0.1
   exit
vlan 1
   name "DEFAULT_VLAN"
   no untagged 1-2
   untagged 3-24, A1-A2, B1-B2
   ip address dhcp-bootp
   exit
vlan 100
   name "OFLAB"
   untagged 1-2
   no ip address
   exit
```

THIS PAGE INTENTIONALLY LEFT BLANK

References

[1] The Cooperative Association for Internet Data Analysis (CAIDA). [Online]. Available: http://www.caida.org/

[2] The Internet Measurement Research Group (IMRG). [Online]. Available: http://www.icir.org/imrg

[3] The Internet Measurement Conference (IMC). [Online]. Available: http://www.sigcomm.org/events/imc-conference

[4] A. Heller, "Defending computer networks against attack," *Lawrence Livermore National Laboratory, Science and Technology Review*, pp. 14–16, January / February 2010.

[5] Open Networking Foundation (ONF). [Online]. Available: https://www.opennetworking.org

[6] OpenFlow. [Online]. Available: http://www.openflow.org/

[7] F. Gens. (2012, November). *IDC Predictions 2013: Competing on the 3rd Platform*. Framingham, MA. [Online]. Available: http://www.idc.com/research/Predictions13/downloadable/238044.pdf

[8] Open Networking Foundation (ONF), "Software-defined networking: The new norm for networks," Open Networking Foundation (ONF), Tech. Rep., April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[9] White House. (2010). National Security Strategy. Washington, DC. [Online]. Available: http://www.whitehouse.gov/sites/default/files/rss_viewer/national_security_strategy.pdf

[10] U.S. Department of Defense. (2011, July). Department of Defense Strategy for Operating in Cyberspace. Washington, DC. [Online]. Available: http://www.defense.gov/news/d20110714cyber.pdf

[11] U.S. Department of Defense. (2010, February). Quadrennial Defense Review Report. Washington, DC. [Online]. Available: http://www.defense.gov/qdr/qdr%20as%20of%2026jan10%200700.pdf

[12] W. Welsh. (2013, February). *All future attacks on the US will include cyber element, says Panetta*. [Online]. Available: http://defensesystems.com/articles/2013/02/06/panetta-speech-future-cyberattacks.aspx

[13] Deputy Directorate, Joint Staff. (2013, June). COMPENDIUM of Key Joint Doctrine Publications. Washington, DC. [Online]. Available: http://www.dtic.mil/doctrine/new_pubs/compendium.pdf

[14] James R. Clapper, Director of National Intelligence. (2013, March). Statement for the Record, Worldwide Threat Assessment of the US Intelligence Community, Senate Select Committee on Intelligence. Washington, DC. [Online]. Available: http://www.dni.gov/files/documents/Intelligence%20Reports/2013%20ATA%20SFR%20for%20SSCI%2012%20Mar%202013.pdf

[15] J. A. Lewis. (2014, February). *Significent Cyber incidents since 2006*. [Online]. Available: http://csis.org/files/publication/131010_Significant_Cyber_Incidents_Since_2006_0.pdf

[16] S. T. Trassare, "A technique for presenting a deceptive dynamic network topology," M.S. thesis, NPS, Monterey, CA, 2013.

[17] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Khan, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, S. Wolff. (2014). *Brief history of the Internet*. [Online]. Available: http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet

[18] Internet world stats, usage and population statistics. [Online]. Available: http://www.internetworldstats.com/

[19] A. Dhamdhere and C. Dovrolis, "The Internet is flat: Modeling the transtition from a transit hierachy to a peering mesh," in *Proceedings of the 6th INternetional Conference*. ACM, 2010.

[20] D. Anderson, "Splinternet behind the great firewall of China," *ACM Queue*, vol. 10, no. 11, 2012.

[21] C. Rhoads and F. Fassihi, "Iran vows to unplug Internet," *The Wall Street Journal*, May 2011.

[22] A. Dainotti, C. Squarcella, E. Aben, K. C. Claffy, M. Chiesa, M. Russo, and A. Pescape, "Analysis of country-wide Internet outages caused by Censorship," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011.

[23] Archipelago measurement infrastructure. [Online]. Available: http://www.caida.org/projects/ark/

[24] Ripe network coordination centre. [Online]. Available: https://www.ripe.net

[25] RIPE Atlas. [Online]. Available: https://atlas.ripe.net/

[26] PlanetLab. [Online]. Available: https://www.planet-lab.org

[27] ANT censuses of the Internet address space. [Online]. Available: http://www.isi.edu/ant/address

[28] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister, "Census and survey of the visible Internet," in *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 2008.

[29] Internet census 2012. [Online]. Available: http://internetcensus2012.bitbucket.org/paper.html

[30] A. Harper, S. Harris, J. Ness, C. Eagle, G. Lenkey, T. Williams, *Gray Hat Hacking: The Ethical Hacker's Handbook*, 3rd ed. New York, NY: McGraw-Hill, 2011, p. 47.

[31] S. McClure, J. Scambray, G. Kurtz, *Hacking Exposed 7: Network Security Secrets and Solutions*. New York, NY: McGraw-Hill, 2012, pp. 31–59.

[32] *A TCP/IP Tutorial*, Internet Engineering Task Force Std. 1180, 1991. [Online]. Available: http://tools.ietf.org/html/rfc1180

[33] *Internet Control Message Protocol*, Internet Engineering Task Force Std. 792, 1981. [Online]. Available: http://tools.ietf.org/html/rfc792

[34] M. J. Muuss. (2014). The story of the PING program. [Online]. Available: http://ftp.arl.army.mil/~mike/ping.html

[35] B. Eriksson, P. Barford, J. Sommers, and R. Nowak, "A learning-based approach for IP geolocation," in *Proceedings of the Thirty-Third Australasian Conferrence on Computer Science*, vol. 102. Australian Computer Society, Inc., 2010.

[36] M. J. Arif, S. Kulkarni, "GeoWeight: Internet host geolocation based on a probability model for latency measurements," in *Passive and Active Measurement*. Springer Berlin Heidelberg, 2010.

[37] Y. Wang, D. Burgener, M. Flores, A. Kuzmanovic, and C. Huang, "Towards street-level client-independent IP geolocation," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI, 2011.

[38] R. Schemers. (2014). fping. [Online]. Available: http://fping.sourceforge.net/

[39] *Requirements for Internet hosts – Communication layers*, Internet Engineering Task Force Std. 1122, 1989. [Online]. Available: http://tools.ietf.org/html/rfc1122

[40] The National Security Agency, *The 60 Minute Network Security Guide*. Fort Meade, MD: The National Security Agency, 2006, p. 17.

[41] S. Sanfilippo. (2014). hping. [Online]. Available: http://www.hping.org

[42] Nping. [Online]. Available: http://nmap.org/nping/

[43] *Internet Protocol*, Internet Engineering Task Force Std. 791, 1981. [Online]. Available: http://tools.ietf.org/html/rfc791

[44] InetDaemon. (2013). *How traceroute works*. [Online]. Available: http://www.inetdaemon.com/tutorials/troubleshooting/tools/traceroute/definations.shtml

[45] Paris traceroute. [Online]. Available: http://www.paris-traceroute.net

[46] Top 125 network security tools. [Online]. Available: http://sectools.org

[47] Nmap. [Online]. Available: http://nmap.org

[48] B. Huffaker, M. Fomenkov, K. C. Claffy, "Internet Topology Data Comparison," CAIDA, University of California, San Diego, CA, USA, Tech. Rep., 2012.

[49] M. Luckie, Y. Hyun, B. Huffaker, "Traceroute probe method and forward IP path inference," in *Internet Measurement Conference (IMC)*, Vouliagmeni, Greece, Oct 2008, pp. 311–324.

[50] The honeynet project. [Online]. Available: http://www.honeynet.org

[51] Project honey pot. [Online]. Available: https://www.projecthoneypot.org

[52] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, H. Zhang, "A clean slate 4D approach to network control and management," in *ACM SIGCOMM Computer Communication Review*, October 2005.

[53] M. Casado and N. McKeown, "The virtual network system," in *Proceedings of the ACM SIGCSE Conference*, 2005, pp. 76–80.

[54] M. Casado, T. Garfinkel, A. Akella, M, Freedman, D. Boneh, N. McKeown, S. Shenker, "SANE: A protection architecture for enterprise networks," in *USENIX Security Symposium*, August 2006.

[55] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *ACM SIGCOMM '07*, Kyoto, Japan, August 2007.

[56] NetFPGA. [Online]. Available: http://netfpga.org

[57] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," in *SIGCOMM Computer Communication Review*, April 2008, pp. 38(2):69–74.

[58] Cisco's one platform kit (onePK). [Online]. Available: http://www.cisco.com/en/US/prod/iosswrel/onepk.html

[59] Juniper Networks JunOS Space SDK. [Online]. Available: http://www.juniper.net/us/en/products-services/network-management/junos-space-sdk/

[60] N. Feamster, J. Rexford, E. Zegura, "The Road to SDN: An Intellectual History of Programable Networks," *ACM Queue*, vol. 11, December 2013.

[61] NOX. [Online]. Available: http://www.noxrepo.org

[62] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," in *ACM SIGCOMM Computer Communication Review*, July 2008, pp. 105–110.

[63] Project floodlight. [Online]. Available: http://www.projectfloodlight.org/

[64] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.0.0, December 2009. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

[65] *OpenFlow Switch Errata*, Open Networking Foundation Std. 1.0.2, October 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.2.pdf

[66] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.3.3, September 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf

[67] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.3.2, April 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf

[68] *VLAN Aggregation for Efficient IP Address Allocation*, Internet Engineering Task Force Std. 3069, 2001. [Online]. Available: http://tools.ietf.org/html/rfc3069

[69] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.1.0, February 2011. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf

[70] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.2, December 2011. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf

[71] *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, Internet Engineering Task Force Std. 2474, 1998. [Online]. Available: http://tools.ietf.org/html/rfc2474

[72] *The Addition of Explicit Congestion Notification (ECN) to IP*, Internet Engineering Task Force Std. 3168, 2001. [Online]. Available: http://tools.ietf.org/html/rfc3168

[73] The netfilter.org project. [Online]. Available: https://www.netfilter.org

[74] *OpenFlow Switch Errata*, Open Networking Foundation Std. 1.0.1, June 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.1.pdf

[75] Ubuntu. [Online]. Available: http://www.ubuntu.com

[76] Graphic Network Simulator (GNS3). [Online]. Available: http://www.gns3.net

[77] BackTrack Linux. [Online]. Available: http://www.backtrack-linux.org

[78] Python. [Online]. Available: http://python.org

[79] GitHub - noxrepo/pox repository. [Online]. Available: https://github.com/noxrepo/pox

[80] POX Wiki. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[81] HP. [Online]. Available: http://www.hp.com

[82] *HP Switch Software OpenFlow Administrator's Guide*, K/KA/WB 15.14, HP, October 2013. [Online]. Available: http://h20628.www2.hp.com/km-ext/kmcsdirect/emr_na-c03991489-1.pdf

[83] H. Newton, *Newton's Telecom Dictionary*, 25th ed. New York, NY: Flatiron Publishing, 2009, pp. 894–895.

[84] K. Keys, Y. Hyun, M. Luckie, and K. Claffy, "Internet-scale IPv4 alias resolution with MIDAR," in *IEEE/ACM Transations on Networking (TON)*, vol. 21, no. 2, 2013, pp. 383–399.

[85] K. Keys, “Internet-scale IP alias resolution techniques,” in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, 2010, pp. 50–55.

[86] Ryu. [Online]. Available: http://osrg.github.io/ryu/

[87] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.4.0, October 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[88] *OpenFlow Switch Specification*, Open Networking Foundation Std. 1.3.0, June 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[89] B. Heller, R. Sherwood and N. McKeown, “The controller placement problem,” in *Proceedings from HotSDN’ 12*. ACM, 2012.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California